I/O-Efficient Shortest Path Algorithms for Undirected Graphs with Random or Bounded Edge Lengths

Ulrich Meyer
Johann-Wolfgang-Goethe-Universität Frankfurt

Norbert Zeh
Dalhousie University

May 18, 2017

Abstract

We present I/O-efficient single-source shortest path algorithms for undirected graphs. Our main result is an algorithm with I/O complexity $O\big(\sqrt{(nm \log L)/B} + \mathrm{MST}(n,m)\big)$ on graphs with n vertices, m edges, and arbitrary edge lengths between 1 and L; $\mathrm{MST}(n,m)$ denotes the I/O complexity of computing a minimum spanning tree; B denotes the disk block size. If the edge lengths are drawn uniformly at random from $(0,1]$, the expected I/O complexity of the algorithm is $O\big(\sqrt{nm/B} + (m/B)\log B + \mathrm{MST}(n,m)\big)$. A simpler algorithm has expected I/O complexity $O\big(\sqrt{(nm \log B)/B} + \mathrm{MST}(n,m)\big)$ for uniformly random edge lengths.

1 Introduction

Given a graph G, a *source vertex* s of G, and an assignment $\ell : E \to \mathbb{R}$ of real *lengths* to the edges of G, the *single-source shortest path (SSSP) problem* is to find the distance $\mathrm{dist}_G(s,x)$ from s to every vertex $x \in G$. This distance is the length of a *shortest path* $\pi(s,x)$ from s to x, where the *length* $\ell(P)$ of a path P is the total length of the edges in P and a shortest path from s to x is a path of minimum length; if no path exists from s to x, we define $\mathrm{dist}_G(s,x) = +\infty$. The notion of a *shortest* path is well-defined only if G contains no negative cycles (cycles whose total edge length is negative). This is guaranteed if all edges in G have non-negative lengths, an assumption that allows the design of faster shortest path algorithms. We also make this assumption in this paper, and we assume G is undirected.

The SSSP problem has been studied extensively, leading to a host of algorithms to solve it. Most of these algorithms use the pointer machine or RAM

(random access machine) model of computation, both of which assume a constant cost per memory access independent of the accessed memory location. Modern computers, however, have memory hierarchies consisting of several levels of cache, main memory, and disks. In such memory hierarchies, the access time for an individual data item may vary by several orders of magnitude, depending on the level in the hierarchy where it is currently stored. It is thus imperative, especially when dealing with large data sets, to design algorithms that are sensitive to the computer's memory hierarchy and try to minimize the number of accesses to slower memory levels.

In this paper, we develop *I/O-efficient* algorithms for the SSSP problem, with the goal of minimizing disk accesses in shortest path computations on graphs beyond the size of the computer's main memory. Assuming a two-level memory hierarchy consisting of main memory and disk keeps our analysis reasonably simple and still leads to results relevant to large-scale computations because main memory and disk are the two memory levels with by far the greatest difference in access times.

We start with a review of previous work and a formal definition of the I/O model of computation; see Section 2. This includes a review of the I/O-efficient breadth-first search (BFS) algorithm by [MM02], which pioneered the ideas on which our algorithms are based. Our shortest path algorithms extend the ideas of [MM02] in two ways. Firstly, we replace the hot pool in the BFS algorithm with a *hierarchy* of hot pools. In Appendix A of the online version, we show that this idea alone, combined with a more efficient priority queue discussed in Section 3, leads to a shortest path algorithm with good average-case performance on arbitrary graphs with uniformly random edge lengths. To obtain a worst-case efficient algorithm, we refine the clustering used in the BFS algorithm and combine it with a more sophisticated criterion where to store edges in the hot pool hierarchy. This algorithm is the main result of our paper and is discussed in Section 4. In Appendix B of the online version, we analyze the average-case performance of this algorithm and prove that it essentially matches that of the BFS algorithm by [MM02]. We conclude with a discussion of open problems and mention some recent related work in Section 5.

2 Preliminaries

2.1 I/O Model

We use the standard *I/O model* with one (logical) disk [AV88] to design and analyze our algorithms. In this model, the computer is equipped with a two-level memory hierarchy consisting of an *internal memory* and a (disk-based) *external memory*. The internal memory is capable of holding M data items (vertices, edges, etc.), while the external memory is of conceptually unlimited size. All computation has to happen on data in internal memory. Data is transferred between internal and external memory in blocks of B consecutive data items. Such a transfer is referred to as an *I/O operation* or *I/O*. The cost of an algorithm

in the I/O model is the number of I/O operations it performs. The number of I/Os needed to read N contiguous items from disk is $\text{scan}(N) = \lceil N/B \rceil$. The number of I/Os required to sort N items is $\text{sort}(N) = \Theta((N/B)\log_{M/B}(N/B))$ [AV88]. For all realistic values of N, M, and B, $\text{scan}(N) < \text{sort}(N) \ll N$.

2.2 Previous Work

The classical shortest path algorithm for graphs with arbitrary edge lengths but no negative cycles is the Bellman-Ford algorithm [FF62, Bel58], whose running time is $O(nm)$. For graphs with non-negative edge lengths, Dijkstra's algorithm [Dij59], when implemented using a Fibonacci heap [FT87], has a running time of $O(n \log n + m)$. These remain the best bounds in the absence of further knowledge about the graph.

If the edge lengths are integers between 1 and L, integer priority queues by [VEBKZ77] and [AMOT90] lead to shortest path algorithms with running times $O(m \log\log L)$ and $O\big(n\sqrt{\log L} + m\big)$, respectively. For undirected graphs with arbitrary integer or floating-point edge lengths, Thorup [Tho99, Tho00] proposed linear-time single-source shortest path algorithms. [PR02] extended Thorup's results to graphs with real edge lengths, achieving a running time of $O(m\alpha(m,n) + n \log\log L)$ on undirected graphs with edge lengths between 1 and L, where $\alpha(\cdot,\cdot)$ is the inverse of Ackermann's function.

All these algorithms use the pointer machine or RAM model of computation, both of which assume a constant cost per memory access independent of the accessed memory location. As pointed out in the introduction, this is far from true in modern computers with hierarchical memory architectures. Consequently, much work has focused on developing algorithms that minimize the number of accesses to slower memory levels, particularly to disk [MSS03, Vit08].

Nearly I/O-optimal algorithms exist for connectivity problems, such as computing connected components or a minimum spanning tree. The best deterministic algorithms for these problems perform $O(\text{sort}(m) \log\log(nB/m))$ I/Os [MR99, ABT04]. The best randomized algorithms perform $O(\text{sort}(m))$ I/Os with high probability [CGG$^+$95, ABW02].

Much less is known about the I/O complexity of graph exploration problems, such as depth-first search (DFS), breadth-first search (BFS), and SSSP. The best bound for these problems on directed graphs is $O((n + m/B)\log n)$ I/Os [KS96, BGVW00], which is also the best DFS bound for undirected graphs. For a long time, the best bounds for BFS and SSSP in undirected graphs were $O(n + \text{sort}(m))$ I/Os [MR99] and $O(n + (m/B)\log(n/B))$ I/Os [KS96], respectively. While these algorithms are efficient on dense graphs, they are only slightly better on sparse graphs than their internal-memory counterparts run in external memory. The challenge is to overcome the "one I/O per vertex" bottleneck in these algorithms.

For undirected BFS, [MM02] achieved this goal and developed an algorithm with I/O complexity $O\big(\sqrt{nm/B} + \text{MST}(n,m)\big)$, where $\text{MST}(n,m)$ denotes the cost of computing a (minimum) spanning tree of the graph. For special graph classes, such as planar graphs and graphs of bounded tree-width, optimal SSSP

algorithms with I/O complexity O(sort(n)) are known [MZ01, ABT04]. [HT06] showed how to extend the results for planar graphs to graphs that are nearly planar for different measures of non-planarity. Recent work by [AMT04] and [CR05] shows how to overcome the vertex bottleneck in all-pairs shortest paths computations; also see Chowdhury's PhD thesis [Cho07].

2.3 The BFS Algorithm by Mehlhorn and Meyer

One of the main challenges with I/O-efficient graph exploration is that the order in which vertices are visited is hard to predict. As a result, naïve graph exploration strategies take at least one random disk access to retrieve the adjacency list of each visited vertex, leading to a cost of O($n + m/B$) I/Os for all adjacency list accesses. The undirected BFS algorithm by [MR99] incurs this cost for retrieving adjacency lists but performs only O(sort(m)) I/Os to construct the BFS tree from the retrieved adjacency lists. Thus, the key to improving on the algorithm's O(n + sort(m)) I/O bound is to reduce the cost of adjacency list accesses. [MM02] proposed the following solution to this problem.

Partition the vertices of G into $q = \mathrm{O}(n/\mu)$ *vertex clusters* $V_1, V_2, \ldots, V_q$ such that any two vertices in the same cluster V_k have distance at most μ from each other; μ is a parameter defined below. Such a partition can be computed using O(MST(n, m)) I/Os by computing a spanning tree of G, partitioning an Euler tour of this tree into $2n/\mu$ subtours of length μ and placing the vertices visited by each subtour into their own vertex cluster V_k; vertices visited by more than one subtour are included in only one vertex cluster, chosen arbitrarily. Now concatenate the adjacency lists of the vertices in each vertex cluster V_k to form a corresponding *edge cluster* E_k and store the edges in each edge cluster consecutively on disk.

After this preprocessing, construct the BFS tree level by level, using a hot pool of edges, H, to speed up adjacency list accesses. Initially, H is empty. Let $L(i)$ be the set of vertices at distance i from s. In particular, $L(0) = \{s\}$. In the ith iteration, construct level $L(i)$ from the two previous levels $L(i-1)$ and $L(i-2)$. First retrieve the adjacency lists of the vertices in $L(i-1)$ to identify the neighbours of all vertices in $L(i-1)$. $L(i)$ is the subset of neighbours that do not belong to $L(i-1)$ or $L(i-2)$ and can be computed exactly as in Munagala and Ranade's algorithm. To retrieve the adjacency lists of the vertices in $L(i-1)$, scan the hot pool H and identify all vertices in $L(i-1)$ whose adjacency lists are *not* in H. Then load the edge clusters containing these adjacency lists into H and scan H a second time to retrieve and remove the adjacency lists of all vertices in $L(i-1)$.

The cost of loading edge clusters into the hot pool is O(n/μ + sort(m)) I/Os because it requires a random access to each edge cluster, followed by sorting the edges in the cluster by their tails and merging them into the hot pool. The hot pool is scanned a constant number of times per iteration of the algorithm. Thus, the contribution of an edge to the scanning cost of the hot pool depends on how long the edge remains in the hot pool. Assume the edge belongs to an adjacency list E_y, is loaded into H as part of an edge cluster E_k in the ith

iteration of the algorithm, and is removed from H in the jth iteration. Then $y \in V_k$, $y \in L(j)$, and the loading of E_k into H was triggered by a vertex $x \in L(i)$ such that $x \in V_k$. Since x and y belong to the same vertex cluster, their distance is at most μ, which implies that, $j - i \leq \mu$. Therefore, every edge contributes at most $O(\mu/B)$ to the scanning cost of the hot pool, and the total scanning cost of the hot pool is $O(\mu m/B)$ I/Os.

Since the cost of the BFS algorithm, excluding the retrieval of adjacency lists, is $O(\text{sort}(m))$ I/Os, the preprocessing takes $O(\text{MST}(n, m))$ I/Os, and the cost of loading edge clusters and manipulating the hot pool is $O(n/\mu + \text{sort}(m) + \mu m/B)$ I/Os, the total cost of the algorithm is $O(n/\mu + \mu m/B + \text{MST}(n, m))$ I/Os. The first two terms in this bound are balanced by choosing $\mu = \sqrt{nB/m}$, which leads to a bound of $O\big(\sqrt{nm/B} + \text{MST}(n, m)\big)$ on the total I/O complexity of the algorithm.

2.4 New Results

We extend the ideas from the BFS algorithm sketched in Section 2.3 in several ways, in order to obtain fast undirected single-source shortest path algorithms.

Our first idea is to refine the hot pool structure used by [MM02] to consist of several levels. The level where an edge is stored depends on its length. This idea, together with an efficient priority queue discussed in Section 3, leads to a simple SSSP algorithm with expected I/O complexity $O\big(\sqrt{(nm \log B)/B} + \text{MST}(n, m)\big)$ on arbitrary undirected graphs with uniformly random edge lengths. This is discussed in Appendix A of the online version of this paper.

By refining the cluster partition and choosing the hot pool where to store each edge more judiciously, we obtain our main result: a worst-case efficient algorithm with I/O complexity $O\big(\sqrt{(nm \log L)/B} + \text{MST}(n, m)\big)$ on undirected graphs with real edge lengths between 1 and L. This algorithm is discussed in Section 4.

In Appendix B of the online version, we provide an average-case analysis of a slight modification of this algorithm, which shows that its expected I/O complexity on arbitrary graphs with uniformly random edge lengths is $O\big(\sqrt{nm/B} + (m/B) \log B + \text{MST}(n, m)\big)$. For sparse graphs, this matches the I/O bound of the BFS algorithm by [MM02] discussed in the previous section.

3 A Batched Priority Queue

Our algorithms are I/O-efficient variants of Dijkstra's algorithm and rely on a priority queue for their implementation. In this section, we discuss the *bucket heap*, a priority queue inspired by the internal-memory priority queue by [AMOT90]. This priority queue is crucial for the efficiency of our algorithms for two reasons. In contrast to other I/O-efficient priority queues [KS96, BFMZ04, CR04], the cost per operation does not depend on the number of elements in the priority queue but on the difference between minimum and maximum priority. In our algorithms, this difference is bounded by the ratio between maximum

and minimum edge length. More importantly, the assignment of vertices to buckets in the bucket heap is a predictor of when a vertex will be visited, and this directly influences where its incident edges need to be stored in the hot pool hierarchy. Also, in order to improve their average-case efficiency, our shortest path algorithms delay edge relaxations, and we use the inspection of buckets to determine how long the relaxation of each edge can be delayed without affecting the correctness of the algorithm.

The bucket heap supports UPDATE(x, p), DELETE(x), and DELETEMIN operations. A DELETE(x) operation removes x from the priority queue. An UPDATE(x, p) operation inserts x into the priority queue, with priority p, if x is currently not in the priority queue; otherwise it replaces x's current priority $p(x)$ with $\min(p, p(x))$. The behaviour of the DELETEMIN operation is relaxed to allow it to return more than one element, as long as the returned elements satisfy the following properties:

(PQ1) The element with minimum priority $p_{\min}$ is among the returned elements.

(PQ2) All returned elements have priority less than $p_{\min} + 1$.

(PQ3) If an element x is returned by the DELETEMIN operation and another element y is left in the priority queue, then $p(x) < p(y)$.

The bucket heap implements these three operations correctly and efficiently if the sequence of UPDATE operations satisfies the following condition.

Condition 1 ((Bounded Update Condition)) *Every* UPDATE(x, p) *operation*
satisfies $p_{max} < p \leq p_{max} + L$, where p_{max} is the maximum priority of all elements removed from the priority queue using DELETEMIN *operations prior to this* UPDATE *operation and L is a parameter chosen in advance. If no elements have been removed using* DELETEMIN *operations before this* UPDATE *operation, $p_{max} = 0$.*

Note that this condition, together with property (PQ3), implies that two elements x and y such that the DELETEMIN operation returning y succeeds the one returning x satisfy $p(x) < p(y)$. In particular, $p_{\max}$ never decreases and $p(x) > p_{\max}$, for all elements x in the priority queue at any point in time.

3.1 Lazy Priority Queues

I/O-efficient priority queues avoid random accesses by performing updates lazily. We can think of such lazy structures as recording, for every element x, the sequence S_x of UPDATE and DELETE operations performed on x. Element x is defined to be *present* in the priority queue if there is at least one UPDATE operation after the last DELETE operation in S_x. The priority of element x is the minimum priority of all UPDATE operations after the last DELETE operation in S_x. A DELETEMIN operation has to satisfy properties (PQ1)–(PQ3) with

respect to all elements present in the priority queue. Moreover, for every returned element x, the DELETEMIN operation has to ensure that x is absent from the priority queue after the operation, which can be achieved by appending a DELETE operation to the end of S_x.

The priority queue may modify the recorded sequence S_x, as long as these modifications do not affect the presence or absence of element x in the priority queue nor its priority. The following rewriting rules have this property. Every internal-memory priority queue can be thought of as recording S_x for all x and, after each operation, applying these rewriting rules to the maximum possible extent.

(RR1) A DELETE operation at the beginning of S_x can be eliminated.

(RR2) Any pair of consecutive operations in S_x, the second one being a DELETE(x) operation, can be replaced with a single DELETE(x) operation.

(RR3) Any pair of consecutive UPDATE operations in S_x, UPDATE(x, p_1) and UPDATE(x, p_2), can be replaced with an UPDATE($x, \min(p_1, p_2)$) operation.

3.2 The Structure

The bucket heap consists of $r+2 = \lfloor \log L \rfloor + 3$ buckets $Q_0, Q_1, \ldots, Q_{r+1}$ storing the operations in sequences S_x, for all elements x. The operations in each bucket are sorted by the elements they affect. For each element x, each bucket contains at most one DELETE(x) and one UPDATE(x, p) operation, with the DELETE operation preceding the UPDATE operation. The only exception to this sortedness condition and this restriction on the number of operations per element is bucket Q_0.

We maintain the invariant that S_x can be obtained by concatenating all operations affecting element x in order of decreasing bucket index and increasing position within each bucket. An UPDATE or DELETE operation simply appends itself to the end of bucket Q_0, thereby maintaining this invariant. Subsequently, the bucket where each operation is stored is determined by *splitter priorities* $q_0 \leq q_1 \leq \cdots \leq q_{r+2} = +\infty$. Initially, we set $q_0 = 0$ and $q_i = 2^{i-1}$, for all $1 \leq i \leq r+1$. Each DELETEMIN operation inspects a subset of the buckets to retrieve the elements to be returned. Each UPDATE(x, p) operation with $q_i \leq p < q_{i+1}$ is stored in the highest bucket Q_j such that $j \leq i$ and Q_{j-1} has been inspected by a DELETEMIN operation since the UPDATE(x, p) operation was inserted into Q_0. Similarly, each DELETE(x) operation is stored in the highest bucket Q_j such that Q_{j-1} has been inspected since its insertion into Q_0.

3.3 DELETEMIN

Since UPDATE(x, p) and DELETE(x) operations simply append themselves to the end of bucket Q_0, the real work is done during DELETEMIN operations.

Each such operation proceeds in two phases. In the *upward phase*, we inspect buckets by increasing index and push operations from each bucket to the next. The inspection of a bucket Q_i moves every DELETE operation in Q_i to Q_{i+1} but moves an UPDATE(x, p) operation to Q_{i+1} only if $p \geq q_{i+1}$. The upward phase ends when we reach a bucket Q_i such that at least one UPDATE operation in Q_i is not moved to Q_{i+1}. In the *downward phase*, we redefine the splitter priorities separating the buckets $Q_0, Q_1, \ldots, Q_i$ inspected during the upward phase and then distribute the UPDATE operations in Q_i over buckets $Q_0, Q_1, \ldots, Q_{i-1}$. At the end of the downward phase, we output all data items corresponding to the UPDATE operations in Q_0. Next we provide the details of these two phases.

Upward phase First we sort the operations in bucket Q_0 by the elements they affect while not altering the order of operations affecting the same element. Thus, the sequence S_x is not changed for any x. Next we eliminate, for each x, all operations affecting x and preceding the last DELETE(x) operation in Q_0. This corresponds to repeated applications of rule (RR2). If there are UPDATE(x, p) operations succeeding this DELETE(x) operation (or there is no DELETE(x) operation in Q_0), we eliminate all of these UPDATE operations, except the one with minimum priority. This corresponds to repeated applications of rule (RR3). Now we repeat the following for increasing bucket indices i, starting with $i = 0$.

If $i = r + 1$, we eliminate all DELETE operations from Q_i, which corresponds to an application of rule (RR1), and then enter the downward phase.

If $i \leq r$, we scan buckets Q_i and Q_{i+1} to move operations from Q_i to Q_{i+1}. For every x, we process the operations in Q_i that affect x in their order of appearance. If Q_i contains a DELETE(x) operation, we remove all operations affecting x from Q_{i+1} and move the DELETE(x) operation from Q_i to Q_{i+1}. By the ordering of operations in Q_i, this corresponds to repeated applications of rule (RR2).

Next, if Q_i contains an UPDATE(x, p) operation, we do the following. If Q_{i+1} contains an UPDATE(x, p') operation, we remove it and replace the UPDATE(x, p) operation in Q_i with an UPDATE(x, p'') operation, where $p'' = \min(p, p')$. This corresponds to an application of rule (RR3). If Q_{i+1} contains no such operation, let $p'' = p$. Now we move the UPDATE(x, p'') operation from Q_i to Q_{i+1} if $p'' \geq q_{i+1}$ or leave it in Q_i if $p'' < q_{i+1}$. In either case, S_x is not altered further.

Once all operations in bucket Q_i have been processed in this manner, Q_i contains no DELETE operations but may contain UPDATE operations. If Q_i is empty, we iteratively apply this procedure to Q_{i+1}. Otherwise we enter the downward phase.

Downward phase Let Q_i be the last bucket inspected in the upward phase. Then Q_i contains only UPDATE operations and buckets $Q_0, Q_1, \ldots, Q_{i-1}$ are empty. The downward phase empties Q_i and distributes its contents over $Q_0, Q_1, \ldots, Q_{i-1}$.

First we update the splitters $q_0, q_1, \ldots, q_i$, which determine where each element in Q_i is sent. To this end, we scan bucket Q_i and identify the UP-

DATE operation with minimum priority $p_{\min}$. We set $q_0 := p_{\min}$ and $q_j := \min(q_{i+1}, p_{\min} + 2^{j-1})$, for all $0 < j \leq i$. Let $h \leq i$ be the highest index such that $q_h \neq q_{i+1}$. If $i < r + 1$, we replace q_h with $(2q_{h-1} + q_{i+1})/3$.

Now we "trickle" UPDATE operations down, by repeating the following procedure for $j = i, i-1, \ldots, 1$: We scan bucket Q_j, remove all UPDATE(x, p) operations with $p < q_j$ from Q_j, and place them into Q_{j-1} without changing their order.

Since $q_0 = p_{\min}$, bucket Q_0 contains at least one operation at the end of this phase unless the priority queue is empty. For each UPDATE(x, p) operation in Q_0, we output element x with priority p and then replace the UPDATE(x, p) operation with a DELETE(x) operation. This corresponds to the mandatory addition of a DELETE(x) operation to the end of S_x after a DELETEMIN operation returns element x, followed immediately by an application of rule (RR2).

3.4 Correctness

To prove the correctness of the bucket heap, we need to verify that every priority queue operation correctly maintains the sequence S_x of the affected element and that a DELETEMIN operation returns only elements that are present in the priority queue and satisfy conditions (PQ1)–(PQ3). To do so, we need the following lemma.

Lemma 1 *After a sequence of priority queue operations such that* UPDATE *operations satisfy the bounded update condition and* DELETEMIN *operations satisfy property (PQ3), every* UPDATE(x, p) *operation in Q_i satisfies $p \geq q_i$, for all $0 \leq i \leq r$.*

Proof 1 *First consider bucket Q_0. Initially, we have $q_0 = 0 = p_{max}$. Subsequently, q_0 is updated only by* DELETEMIN *operations. Each such operation changes q_0 to the priority of an element returned by the* DELETEMIN *operation. Hence, we have $q_0 \leq p_{max}$ at all times. On the other hand, we have $p \geq p_{max}$, for all* UPDATE(x, p) *operations in the priority queue, by property (PQ3) and the bounded update condition. Thus, $p \geq q_0$, for all* UPDATE(x, p) *operations in Q_0.*

For a bucket Q_i with $i > 0$, the upward phase of a DELETEMIN *operation propagates an* UPDATE(x, p) *operation from Q_{i-1} to Q_i only if $p \geq q_i$. The downward phase redefines some set of splitters $q_0, q_1, \ldots, q_i$ and then distributes the operations in Q_i over buckets $Q_0, Q_1, \ldots, Q_{i-1}$ so that every* UPDATE(x, p) *operation in Q_j satisfies $q_j \leq p < q_{j+1}$.*

The following lemma states invariants of the splitter priorities $q_0, q_1, \ldots, q_{r+2}$. Its proof if straightforward and therefore omitted.

Lemma 2 *The splitter priorities $q_0, q_1, \ldots, q_{r+2}$ satisfy the following invariants:*

(i) $0 < q_1 - q_0 \leq 1$,

(ii) For $1 \leq i \leq r$, either $q_{i+1} = q_i$ or $2^{i-1}/3 \leq q_{i+1} - q_i \leq 2^{i-1}$, and

(iii) $q_{r+2} = +\infty$.

Using Lemmas 1 and 2, we can now prove the correctness of the bucket heap.

Lemma 3 *The bucket heap is a correct priority queue, provided the sequence of* Update *operations satisfies the bounded update condition.*

Proof 2 *We have already argued that* Update *and* Delete *operations correctly extend S_x by appending themselves to the end of this sequence, and that all modifications of the buckets made by* DeleteMin *operations correspond to applications of rules (RR1)–(RR3). A* DeleteMin *operation correctly eliminates every returned element x from the priority queue by appending a* Delete(x) *operation to the end of S_x. For every element x returned by a* DeleteMin *operation, the* Update(x, p) *operation in Q_0 is the last operation in S_x. Hence, x is present in the priority queue at the time it is returned. Thus, it suffices to prove that p is the correct priority of element x and that the set of returned elements satisfies conditions (PQ1)–(PQ3). Assume by induction that the* DeleteMin *operations preceding the current* DeleteMin *operation satisfy conditions (PQ1)–(PQ3).*

The priority of an element x such that Q_0 contains an Update(x, p) *operation cannot be greater than p because this* Update(x, p) *operation is the last operation in S_x. Any other* Update(x, p') *operation is stored in a bucket Q_j with $j \geq 1$ and, thus, by Lemma 1, has priority $p' \geq q_1 > p$. Hence, the priority of x is exactly p.*

Every element y that is present in the priority queue and has no Update(y, p') *operation in Q_0 is represented by* Update(y, p') *operations in buckets starting with Q_1 and, thus, has priority at least $q_1 > p$. This proves properties (PQ1) and (PQ3).*

To prove property (PQ2), observe that all returned elements have priorities between q_0 and q_1, where the comparison with q_1 is strict. Thus, since $q_1 - q_0 \leq 1$ (Lemma 2(i)), every returned element has priority less than $p_{min} + 1$.

3.5 I/O Complexity

To analyze the I/O complexity of a sequence of operations on the bucket heap, we need the following observation and its corollary. It implies that every DeleteMin operation that scans buckets $Q_0, Q_1, \ldots, Q_i$ leaves bucket Q_i empty and, thus, forces the next DeleteMin operation that scans Q_i to also scan bucket Q_{i+1}.

Observation 1 *If a* DeleteMin *operation scans buckets $Q_0, Q_1, \ldots, Q_i$ with $0 < i \leq r$, then $q_i = q_{i+1}$ after this operation.*

Corollary 1 *If a* DeleteMin *operation scans buckets $Q_0, Q_1, \ldots, Q_i$, where $i > 0$, each* Update(x, p) *operation in Q_i is moved to Q_{i+1} if $p \geq q_{i+1}$ or to a bucket Q_j with $j < i$ if $p < q_{i+1}$. Once an* Update(x, p) *operation moves from Q_i to a bucket Q_j with $j < i$, it never moves to a bucket Q_h with $h > j$ again.*

Proof 3 *If $p \geq q_{i+1}$, the upward phase of the* DeleteMin *operation moves the* Update(x, p) *operation to Q_{i+1}. If $p < q_{i+1}$, the downward phase moves the operation to Q_{i-1} (and possibly lower) because, as we argue next, $p < q_i$. If $i \leq r$, this follows because $p < q_{i+1}$ and, by Observation 1, $q_i = q_{i+1}$ after updating splitters. If $i = r+1$, observe that we set $q_j = p_{min} + 2^{j-1}$, for all $1 \leq j \leq r+1$, in this case. Hence, $q_{r+1} = p_{min} + 2^r = p_{min} + 2^{\lfloor \log L \rfloor + 1} > p_{min} + L$. As observed on page 6, the bounded update condition implies that $p_{min} > p_{max}$ and every* Update(x, p) *operation in the priority queue satisfies $p \leq p_{max} + L$. Hence, we have $p < q_{r+1}$, for every* Update(x, p) *operation in Q_{r+1}.*

An Update(x, p) *operation that moves to a bucket Q_j with $j < i$ satisfies $q_j \leq p < q_{j+1}$. Since the priority of this operation does not change, it can move to bucket Q_{j+1} only after decreasing q_{j+1} to a value no greater than p, which requires bucket Q_j to become empty first. For Q_j to become empty, the* Update(x, p) *operation has to be removed from Q_j, either by moving it to a lower bucket or by deleting it. In either case, the operation does not move to Q_{j+1} again.*

It is easy to see that, with $B(r+2) = \mathrm{O}(B \log L)$ main memory, we can afford to keep one block per bucket in main memory. This allows us to scan each bucket without first performing a random I/O to access the first block in the bucket, which is crucial for the proof of the next lemma. Since we always inspect a prefix $Q_0, Q_1, \ldots, Q_i$ of the bucket hierarchy, however, we can reduce the main memory requirements to $M = \mathrm{O}(B)$ by storing the buckets on a stack and implementing the scanning of buckets using stack operations.

Lemma 4 *A sequence of N* Update, Delete, *and* DeleteMin *operations on a bucket heap can be processed using $\mathrm{O}(\mathrm{sort}(N) + (N/B) \log L)$ I/Os.*

Proof 4 *Every* Update *or* Delete *operation appends itself to bucket Q_0, while a* DeleteMin *operation inserts one* Delete *operation per returned element. Since every returned element can be charged to the* Update *operation that inserted it into the priority queue, the total number of insertions of operations into bucket Q_0 is at most $2N$. Appending these operations to bucket Q_0 takes $\mathrm{O}(N/B)$ I/Os.*

The total cost of sorting and compacting bucket Q_0 during DeleteMin *operations is $\mathrm{O}(\mathrm{sort}(N))$ I/Os. Indeed, the downward phase of a* DeleteMin *operation leaves Q_0 empty, except for the* Delete *operations affecting the returned elements. Hence, every operation in Q_0 at the beginning of a* DeleteMin *operation was inserted into Q_0 as part of or after the downward phase of the previous* DeleteMin *operation. Thus, every operation is involved in exactly one sorting and compaction step, and the total amount of data that is sorted and scanned is at most $2N$.*

Moving operations between buckets involves scanning adjacent buckets. We prove that every operation is scanned only $\mathrm{O}(1)$ times per level. Thus, since there are $\mathrm{O}(N)$ operations and $\mathrm{O}(\log L)$ levels, the cost of moving elements between buckets is $\mathrm{O}((N/B) \log L)$ I/Os. There are three types of accesses to

an operation: (1) as part of Q_i when moving operations from Q_i to Q_{i+1} in an upward phase, (2) as part of Q_{i+1} when moving operations from Q_i to Q_{i+1} in an upward phase, and (3) as part of Q_i when moving operations from Q_i to Q_{i-1} in a downward phase.

Accesses of type (1) happen at most twice per operation and level. Indeed, by Corollary 1, we can split the lifetime of every Update *operation into two stages. As long as it has not reached the highest bucket it will ever be stored in during its lifetime, we say the operation* ascends. *As soon as it reaches this highest bucket, it starts to* descend. Delete *operations only ascend. While an operation ascends, every type (1) access moves it to the next bucket; this accounts for one type (1) access per level. Whenever a type (1) access to the operation is made while it descends, the bucket Q_i containing it is the last bucket scanned by the* DeleteMin *operation because the operation is not moved to Q_{i+1} and, hence, Q_i is non-empty after this scan. The following downward phase moves the operation to Q_{i-1} or lower. Thus, there cannot be another type (1) access to this operation in Q_i.*

Type (3) accesses happen only while an operation descends. Let Q_{i_0} be the bucket where the operation starts descending, and let $Q_{i_1}, Q_{i_2}, \ldots, Q_{i_h}$ be the buckets containing the operation at the ends of the DeleteMin *operations that make type (3) accesses to it. Then, by Corollary 1, $i_0 > i_1 > \cdots > i_h$, and the* DeleteMin *operation moving the operation from bucket Q_{i_j} to bucket $Q_{i_{j+1}}$ scans the operation once in each of the buckets $Q_{i_j}, Q_{i_j-1}, \ldots, Q_{i_{j+1}}$. Thus, there are at most two type (3) accesses per operation and level.*

Finally, observe that the number of type (2) accesses is at most twice the number of type (1) accesses: Consider an operation in bucket Q_{i+1}. If the upward phase of a DeleteMin *operation scans up to a bucket Q_j, $j > i$, then the type (2) access to Q_{i+1} is followed immediately by a type (1) access. If Q_i is the last bucket scanned by the upward phase, then, by Observation 1, the priority interval of Q_i becomes empty and remains empty until operations from a higher bucket Q_j, $j > i$, are distributed over buckets $Q_0, Q_1, \ldots, Q_{j-1}$. Thus, the next time a type (2) access is made to Q_{i+1}, Q_i remains empty, and the upward phase of the* DeleteMin *operation does continue to Q_{i+1}, resulting in a type (1) access to Q_{i+1}.*

Lemmas 3 and 4 together prove the following result.

Theorem 1 *The bucket heap is a priority queue that processes a sequence of N* Update, Delete, *and* DeleteMin *operations using $\mathrm{O}(\mathrm{sort}(N)+(N/B)\log L)$ I/Os, provided the sequence satisfies the bounded update condition with parameter L.*

The final lemma of this section shows that the frequency of accesses to a bucket Q_i decreases exponentially with increasing i, which is crucial for the analysis of the shortest path algorithm in the next section. In this lemma and in the rest of the paper, we say a DeleteMin operation *inspects level i* if the upward phase of this operation scans buckets Q_i and Q_{i+1} to propagate operations from Q_i to Q_{i+1}.

Lemma 5 *Consider two* DeleteMin *operations o_1 and o_2 that both inspect a level $i \geq 1$, assume o_1 occurs before o_2, and let p_1 and p_2 be the minimum priorities of the elements returned by o_1 and o_2, respectively. The* DeleteMin *operations between o_1 and o_2, inclusive, inspect level i at most $2+3(p_2-p_1)/2^{i-1}$ times.*

Proof 5 *Let $o_1 = o'_1, o'_2, \ldots, o'_k = o_2$ be the sequence of* DeleteMin *operations between o_1 and o_2 that inspect level i. For each operation o'_j, let $\ell(j)$ be the highest level inspected by operation o'_j, q'_j the value of $q_{\ell(j)}$ immediately before operation o'_j, and p'_j the minimum priority of all elements returned by operation o'_j. Then $p'_j \geq q'_j$. Hence, it suffices to prove that $q'_j \geq p_1 + (j-2)2^{i-1}/3$. Moreover, it suffices to prove this claim for $j > 1$ because there is nothing to prove if $k = 1$.*

Consider first the case $i \leq r$. We prove the claim by induction on j. For $j = 2$, we have to prove that $q'_2 \geq p_1$. By Observation 1, operation o'_1 leaves the priority interval of bucket $Q_{\ell(1)}$ empty; therefore, $\ell(2) \neq \ell(1)$. If $\ell(2) > \ell(1)$, we have $p_1 = p'_1 < q_{\ell(1)+1} \leq q_{\ell(2)} = q'_2$. If $\ell(2) < \ell(1)$, then $q'_2 > p_1$ because operation o'_1 computes $q_1, q_2, \ldots, q_{\ell(1)}$ by adding positive values to p_1.

Now assume $j > 2$ and $q'_{j-1} \geq p_1 + (j-3)2^{i-1}/3$. Again, we have $\ell(j) \neq \ell(j-1)$ because operation o'_{j-1} leaves the priority interval of bucket $Q_{\ell(j-1)}$ empty. If $\ell(j) > \ell(j-1)$, we have $q'_j = q_{\ell(j)} \geq q_{\ell(j-1)+1} \geq q_{\ell(j-1)} + 2^{\ell(j-1)-1}/3 \geq q'_{j-1} + 2^{i-1}/3 \geq p_1 + (j-2)2^{i-1}/3$, by Lemma 2(ii) and because $\ell(j-1) \geq i \geq 1$.

If $\ell(j) < \ell(j-1)$, then operation o'_{j-1} must have assigned non-empty priority intervals to buckets $Q_0, Q_1, \ldots, Q_{\ell(j)}$, because bucket $Q_{\ell(j)}$ is non-empty at the time of operation o'_j. Hence, $q'_j = q_{\ell(j)} \geq p'_{j-1} + 1/3 + \sum_{h=1}^{\ell(j)-1} 2^{h-1}/3 \geq q'_{j-1} + 1/3 + \sum_{h=0}^{\ell(j)-2} 2^h/3 \geq q'_{j-1} + 2^{\ell(j)-1}/3 \geq q'_{j-1} + 2^{i-1}/3$, as in the case $\ell(j) > \ell(j-1)$. So the inductive claim holds in this case as well.

Finally, if $i = r+1$, observe that operations $o'_1, o'_2, \ldots, o'_k$ all scan bucket Q_{r+1}. Operation o'_j sets $q_{r+1} = p'_j + 2^r \geq q'_j + 2^r$. Hence, $q'_{j+1} \geq q'_j + 2^r$, for all $1 \leq j < q$, that is, $q'_{j+1} \geq p_1 + j2^r$.

4 A Worst-Case Efficient Algorithm for Bounded Edge Lengths

In this section, we prove our main result:

Theorem 2 *Let G be an undirected graph with n vertices, m edges, and real edge lengths between 1 and L. The single-source shortest path problem on G can be solved using $\mathrm{O}\Big(\sqrt{(nm\log L)/B} + \mathrm{MST}(n,m)\Big)$ I/Os.*

To obtain this result, we require the bucket heap from the previous section, a hot pool hierarchy whose manipulation will be closely tied to the manipulation of priority queue buckets, and a cluster partition that takes edge lengths into account. We describe the cluster partition in Section 4.1. In Section 4.2, we

define *cluster trees* and present an algorithm to compute them efficiently. These trees encode the internal structure of the clusters in the cluster partition and provide the information necessary to choose the hot pool where to store each edge. In Section 4.3, we discuss the hot pool hierarchy and the shortest path algorithm.

4.1 A Weighted Cluster Partition

Our algorithm uses a similar cluster partition as the linear-time shortest path algorithms for undirected graphs with integer and floating-point edge lengths by Thorup [Tho99, Tho00]. For $0 \leq i \leq r = \lfloor \log L \rfloor + 1$, let G_i be the subgraph of G that contains all vertices of G and all edges in categories $1, 2, \ldots, i$, where a *category-i edge* e satisfies $2^{i-1} \leq \ell(e) < 2^i$. The category of a vertex is the minimum category of its incident edges. We call a connected component of G_i an *i-component* of G and denote the i-component containing a vertex x by $[x]_i$. Now fix some parameter $3 \leq \mu \leq \sqrt{B}$. An *i-cluster* is a vertex set V' with the following properties.

(C1) There exists an i-component containing all vertices in V'.

(C2) Any two vertices in $(i-1)$-components that are non-disjoint from V' have distance at most $\mu 2^i$ from each other in G.

(C3) For every vertex $x \in V'$ and every $j < i$, the j-component $[x]_j$ has diameter less than $\mu 2^j$.

We call i the *category* of cluster V'. We call a partition of G into vertex clusters $V_1, V_2, \ldots, V_q$ *proper* if it satisfies the following three conditions.

(PP1) $q = \mathrm{O}(n/\mu)$.

(PP2) Every cluster V_k is an i-cluster, for some i.

(PP3) For every i-component $[x]_i$, either $j \leq i$, for every j-cluster containing a vertex from $[x]_i$, or there exists at most one j-cluster that contains vertices from $[x]_i$ and such that $j \geq i$.

In this section, we show how to compute a proper cluster partition of G efficiently:

Theorem 3 *Given an undirected graph G with n vertices, m edges, and edge lengths between 1 and L, as well as a parameter $3 \leq \mu \leq \sqrt{B}$, a proper cluster partition of G can be computed using $\mathrm{O}(\mathrm{MST}(n, m) + (n/B) \log L)$ I/Os.*

By the following observation, it suffices to compute a proper cluster partition of a minimum spanning tree T of G.

Observation 2 *A proper cluster partition of a minimum spanning tree T of G is also a proper cluster partition of G.*

To compute a proper cluster partition of T, we iterate over categories $i = 1, 2, \ldots, r$ and, for each i, partition the i-components of T into i-clusters.

First we compute an *Euler tour* $\mathcal{E}([x]_i)$ for each i-component $[x]_i$. $\mathcal{E}([x]_i)$ is a list of directed edges yz such that, for every undirected edge $yz \in [x]_i$, there exists an edge yz and an edge zy in $\mathcal{E}([x]_i)$ and, for any two consecutive edges in $\mathcal{E}([x]_i)$, the head of the first is the tail of the second. The length of an edge in $\mathcal{E}([x]_i)$ equals the length of its corresponding edge in $[x]_i$.

Now assume every vertex in T stores whether it has been added to a vertex cluster in a previous iteration. If all vertices in $[x]_i$ have been assigned to vertex clusters or $\mathcal{E}([x]_i)$ has length less than $\mu 2^i$ and $i < r$, we do not partition $[x]_i$ into i-clusters. Otherwise we split $\mathcal{E}([x]_i)$ into subtours $\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_t$, each of which defines an i-cluster. We inspect the edges in $\mathcal{E}([x]_i)$ in order, adding them to $\mathcal{E}_1$ as long as the length of $\mathcal{E}_1$ does not exceed $\mu 2^{i-1}$. As soon as the addition of the next edge would make the length of $\mathcal{E}_1$ exceed $\mu 2^{i-1}$, we make this edge the first edge in a new subtour $\mathcal{E}_2$ and continue adding edges to $\mathcal{E}_2$ until its length is about to exceed $\mu 2^{i-1}$, and so on. For each $(i-1)$-component $[y]_{i-1}$, let h be the minimum index such that $\mathcal{E}_h$ visits a vertex in $[y]_{i-1}$. We assign every vertex in $[y]_{i-1}$ that has not been assigned to a vertex cluster yet to the vertex cluster corresponding to subtour $\mathcal{E}_h$ and mark it as added to a cluster to prevent its addition to any other cluster in a subsequent iteration. Before discussing how to implement this procedure I/O-efficiently, we prove that the resulting cluster partition is proper.

Lemma 6 *The above procedure computes a proper cluster partition of T (and, hence, of G).*

Proof 6 *First we prove that the computed clusters satisfy condition (PP2). More specifically, we prove that every cluster produced in the ith iteration is an i-cluster. Each such cluster V_k is contained in an i-component $[x]_i$, that is, it satisfies condition (C1). To verify condition (C2), consider two vertices y and z such that $[y]_{i-1} = [y']_{i-1}$ and $[z]_{i-1} = [z']_{i-1}$, for two vertices y' and z' in V_k. Since $y', z' \in V_k$, there exist two vertices $y'' \in [y]_{i-1}$ and $z'' \in [z]_{i-1}$ that are both visited by the subtour $\mathcal{E}_h$ of $\mathcal{E}([x]_i)$ that defines V_k. The length of $\mathcal{E}_h$ is at most $\mu 2^{i-1}$, so $\mathrm{dist}_T(y'', z'') \leq \mu 2^{i-1}$. Since both $[y]_{i-1}$ and $[z]_{i-1}$ contain vertices that are unmarked at the beginning of the ith iteration—namely y' and z'—we did not partition $[y]_{i-1}$ or $[z]_{i-1}$ into clusters in the $(i-1)$st iteration. This implies that $\mathcal{E}([y]_{i-1})$ and $\mathcal{E}([z]_{i-1})$ have length at most $\mu 2^{i-1}$, that is, $[y]_{i-1}$ and $[z]_{i-1}$ have diameter at most $\mu 2^{i-2}$ and $\mathrm{dist}_T(y, y'') \leq \mu 2^{i-2}$ and $\mathrm{dist}_T(z'', z) \leq \mu 2^{i-2}$. By summing these distances, we obtain $\mathrm{dist}_T(y, z) \leq \mu 2^i$. Finally, to verify condition (C3), observe that every j-component $[x]_j$ such that $\mathcal{E}([x]_j)$ has length at least $\mu 2^j$ is partitioned into clusters in the jth iteration. Thus, if $[x]_j$ contributes vertices to an i-cluster with $i > j$, $\mathcal{E}([x]_j)$ has length less than $\mu 2^j$ and, hence, the diameter of $[x]_j$ is less than $\mu 2^j$.*

Next we verify condition (PP3). If all vertices in an i-component $[x]_i$ are assigned to clusters by the end of the ith iteration, all clusters containing vertices from $[x]_i$ are j-clusters with $j \leq i$. Otherwise we cannot have partitioned $[x]_i$

into clusters in the ith iteration, that is, all clusters that contain vertices in $[x]_i$ and were formed during the first i iterations are j-clusters with $j < i$. Now consider the minimum $j > i$ such that $\mathcal{E}([x]_j)$ has length at least $\mu 2^j$ or $j = r$. Then all unassigned vertices in $[x]_j$ are assigned to clusters in the jth iteration, and the construction explicitly ensures that the unassigned vertices in $[x]_{j-1} \supseteq [x]_i$ are placed into the same cluster. Thus, $[x]_i$ satisfies condition (PP3).

To verify condition (PP1), let m_i denote the number of category-i edges, and q_i the number of clusters produced in the ith iteration. The total length of the Euler tours of all i-components is $\ell_i < 2\sum_{j=1}^{i} 2^j m_j$. If $i < r$, only i-components whose Euler tours have length at least $\mu 2^i$ are partitioned into clusters; thus, there are $c_i \leq \ell_i/(\mu 2^i)$ i-components that are partitioned into clusters. For $i = r$, we have $c_r = 1$. Each cluster, except the last one in each component, corresponds to a subtour $\mathcal{E}_h$ of length at least $(\mu-2)2^{i-1}$. Indeed, adding another edge e to $\mathcal{E}_h$ would produce a subtour of length greater than $\mu 2^{i-1}$. Since e is in a category $j \leq i$, its length is less than 2^i, which implies that the length of $\mathcal{E}_h$ is at least $(\mu-2)2^{i-1}$. This implies that each i-component whose Euler tour has length $\ell \geq \mu 2^i$ is partitioned into $\lceil \ell/((\mu-2)2^{i-1})\rceil \leq \ell/((\mu-2)2^{i-1}) + 1$ clusters. By summing over all such i-components, we obtain that, for $i < r$, the total number of i-clusters is

$$q_i \leq c_i + \frac{\ell_i}{(\mu-2)2^{i-1}} \leq \frac{3\ell_i}{(\mu-2)2^i} \leq \frac{6}{(\mu-2)2^i}\sum_{j=1}^{i} 2^j m_j.$$

For $i = r$, the number of r-clusters is

$$q_r \leq 1 + \frac{\ell_r}{(\mu-2)2^{r-1}} \leq 1 + \frac{3\ell_r}{(\mu-2)2^r} \leq 1 + \frac{6}{(\mu-2)2^r}\sum_{j=1}^{r} 2^j m_j.$$

By summing over all i, we obtain that the total number of clusters is at most

$$\begin{aligned} 1 + \sum_{i=1}^{r}\left(\frac{6}{(\mu-2)2^i}\sum_{j=1}^{i} 2^j m_j\right) &= 1 + \sum_{j=1}^{r}\left(\frac{6m_j}{\mu-2}\sum_{i=0}^{r-j} 2^{-i}\right) \leq 1 + \sum_{j=1}^{r}\frac{12m_j}{\mu-2} \\ &= 1 + \frac{12(n-1)}{\mu-2} = \mathrm{O}\left(\frac{n}{\mu}\right). \end{aligned}$$

To prove Theorem 3, it remains to show how to implement the clustering procedure I/O-efficiently. We compute a minimum spanning tree T of G and an Euler tour $\mathcal{E}(T)$ of T. This takes $\mathrm{O}(\mathrm{MST}(n,m))$ and $\mathrm{O}(\mathrm{sort}(n))$ I/Os, respectively [ABT04, CGG$^+$95]. By sorting and scanning $\mathcal{E}(T)$, we identify the first occurrence of every vertex in $\mathcal{E}(T)$ and label it as unassigned to a cluster. Next we proceed in r iterations. In the ith iteration, we scan $\mathcal{E}(T)$ twice and form i-clusters.

In both scans, we distinguish between *forward* and *backward* edges pointing away from and toward the start vertex of $\mathcal{E}(T)$ in T, respectively. To do so, we maintain a stack E that stores all edges of T we have traversed only once so

far. In a forward scan of $\mathcal{E}(T)$, if the edge yx corresponding to the current edge xy is not on the top of stack E, then xy is a forward edge, and we push it onto E. If edge yx is on the top of stack E, then xy is a backward edge, and we pop edge yx from E. In a backward scan of $\mathcal{E}(T)$, the roles of forward and backward edges are reversed. Our discussion of the two scans of $\mathcal{E}(T)$ below does not include the maintenance of stack E, but both scans update E to distinguish between the two types of edges.

In the first scan of the ith iteration, we scan $\mathcal{E}(T)$ backward and label the first vertex in each i-component $[x]_i$ visited by $\mathcal{E}(T)$ with the total length of $\mathcal{E}([x]_i)$. To this end, we maintain a stack L that stores the current length of $\mathcal{E}([x]_i)$, for every partially traversed i-component $[x]_i$. Initially, L stores a length of 0 to represent the i-component containing the last vertex of $\mathcal{E}(T)$ (which is also the first vertex of $\mathcal{E}(T)$). As we scan $\mathcal{E}(T)$ backward, we do the following for every edge xy.

— If the category of edge xy is at most i, we add $\ell(xy)$ to the top entry of stack L, as edge xy contributes to the length of $\mathcal{E}([x]_i)$.

— If the category of edge xy is greater than i, then $[x]_i \neq [y]_i$. If xy is a forward edge, we are done traversing $[y]_i$ and the top entry of stack L stores the length of $\mathcal{E}([y]_i)$. We pop this entry from L and store it with edge xy. If xy is a backward edge, we are entering a new i-component $[x]_i$ and push a new entry, initially 0, onto L.

After all edges of $\mathcal{E}(T)$ have been inspected, L contains one entry, which represents the length of the Euler tour of the i-component containing the start vertex of $\mathcal{E}(T)$. We store this length with this start vertex.

The second scan scans $\mathcal{E}(T)$ forward and partitions the i-components of T into i-clusters. We maintain three stacks: L, R, and C. L stores, for each partially traversed i-component, the length of the traversed portion of its Euler tour. R stores the ID of the most recently created cluster in each such component. C stores the IDs of the clusters to which the unassigned vertices in partially traversed $(i-1)$-components are to be assigned. Initially, all three stacks are empty. Next we push entries representing the i-component $[x]_i$ containing the start vertex x of $\mathcal{E}(T)$ onto these three stacks. We do this by pretending to traverse a category-∞ forward edge with head x and processing it as described below. Then we process the edges of $\mathcal{E}(T)$ in order. The actions we take for each edge xy in $\mathcal{E}(T)$ depend on its category.

— If the category of edge xy is greater than i, then $[x]_i \neq [y]_i$.

  If xy is a forward edge, y is the first vertex of $[y]_i$ we visit and edge xy stores the length of $\mathcal{E}([y]_i)$. If $\mathcal{E}([y]_i)$ has length at least $\mu 2^i$ ($[y]_i$ is to be partitioned into i-clusters), we create a new cluster ID k and push it onto stacks C and R to indicate that all unassigned vertices in $[y]_{i-1}$ are to be assigned to cluster V_k. We also push a 0 onto stack L. If vertex y has not been assigned to a cluster yet, we add a pair (y, k) to a list P to indicate

that $y \in V_k$ and mark vertex y as assigned to a cluster. If $\mathcal{E}([y]_i)$ has length less than $\mu 2^i$, we push NIL onto stack C to indicate that $[y]_i$ is not to be partitioned into clusters. Nothing is pushed onto L or R in this case.

If xy is a backward edge, we are done traversing $[x]_i$, and we remove the topmost entry from stack C. If this entry is non-NIL, we also remove the topmost entries from stacks L and R.

— If the category of edge xy is i, then $[x]_i = [y]_i$ but $[x]_{i-1} \neq [y]_{i-1}$. If the top of stack C is a cluster ID $k \neq$ NIL ($[x]_i$ is to be partitioned into i-clusters), we add $\ell(xy)$ to the top entry of stack L. If xy is a backward edge, we pop the top entry from C because we have finished traversing the $(i-1)$-component $[x]_{i-1}$ corresponding to this entry. Otherwise we check whether the top entry of L is greater than $2^{i-1}\mu$. If so, we push a new cluster ID onto C to assign all unassigned vertices in $[y]_{i-1}$ to a new cluster, replace the top of R with this cluster ID, and reset the top entry of L to 0. Otherwise we push the cluster ID on the top of R onto C to indicate that the most recently created i-cluster in $[x]_i$ is to include all unassigned vertices in $[y]_{i-1}$. Finally, if vertex y has not been assigned to a cluster yet, we add an entry (y, k) to list P to assign y to cluster V_k, where k is the top entry of C, and mark y as assigned to a cluster.

— If the category of edge xy is less than i, then $[x]_{i-1} = [y]_{i-1}$, and the top entry k of stack C is the index of the cluster V_k to which the vertices of $[x]_{i-1}$ are to be assigned. If $k \neq$ NIL, we add $\ell(xy)$ to the topmost entry on stack L. If in addition xy is a forward edge and vertex y has not been assigned to a cluster yet, we add the pair (y, k) to list P and mark y as added to a cluster.

After we have processed $\mathcal{E}(T)$ in this manner, we sort the entries in P by their second components, thereby arranging the vertices in each cluster consecutively.

It is easily verified that this procedure has the I/O complexity stated in Theorem 3 because it computes a minimum spanning tree T and its Euler tour $\mathcal{E}(T)$, and scans $\mathcal{E}(T)$ $O(\log L)$ times. During each scan, every edge gives rise to a constant number of stack operations, so their cost is bounded by the cost of scanning $\mathcal{E}(T)$. The cost of sorting P is dominated by the cost of the MST computation.

It is equally easy to verify that the above procedure is a faithful implementation of the clustering procedure leading up to Lemma 6, except that it does not start a new subtour $\mathcal{E}_h$ of the Euler tour $\mathcal{E}([x]_i)$ of an i-component $[x]_i$ until a new $(i-1)$-component is entered. This can clearly only reduce the number of clusters. All $(i-1)$-components traversed between the time the length of $\mathcal{E}_{h-1}$ exceeds $\mu 2^{i-1}$ and the start of $\mathcal{E}_h$ have already been assigned to clusters based on the part of $\mathcal{E}([x]_i)$ traversed before the length of $\mathcal{E}_{h-1}$ exceeded $\mu 2^{i-1}$. Thus, the diameter of each of these clusters is unaffected by the delay in starting a new subtour $\mathcal{E}_h$, and we obtain a proper cluster partition of G; Theorem 3 is proved.

Our shortest path algorithm also requires the edge clusters corresponding to the computed vertex clusters. These edge clusters can be computed using O(sort(m)) I/Os: Sort the elements of P by their first components (vertex IDs), and sort the edges in the concatenation of all adjacency lists by their tails. A single scan of these two sorted lists suffices to label every edge xy with the ID of the cluster containing vertex x. Finally, we form edge clusters by sorting the edges by their cluster IDs.

4.2 Cluster Trees

Each vertex cluster's cluster tree encodes how this cluster interacts with the i-components of G. This information is used to decide where to store each edge in the corresponding edge cluster in the hot pool hierarchy. We call an i-component $[x]_i$ of G *minimal* if $i = 0$ or $[x]_i$ consists of at least two $(i-1)$-components. The *component tree* $\mathcal{C}(G)$ of G is a rooted tree whose nodes are the minimal i-components of G, for all i. A minimal i-component $[x]_i$ is a child of a minimal j-component $[x]_j$ if $i < j$ and all components $[x]_h$ with $i < h < j$ are non-minimal, that is, $[x]_i = [x]_h$, for each such component $[x]_h$. The *cluster tree* $\mathcal{C}_k$ of an i-cluster V_k is the subtree of $\mathcal{C}(G)$ containing all nodes $[x]_j$ such that $x \in V_k$ and $j \leq i$.

We compute cluster trees in three steps: First we compute the component tree $\mathcal{C}(G)$ of G, then we assign contiguous vertex IDs to the vertices in each i-component, and finally we process $\mathcal{C}(G)$ bottom-up to extract the cluster trees.

Computing the component tree To compute the component tree of G, we use an algorithm by [ATZ03], which extracts $\mathcal{C}(G)$ from a minimum spanning tree of G using O(sort(n)) I/Os. More recently, [AAY06] proposed a simpler procedure with the same I/O complexity for a slightly different problem, but which can easily be adapted to compute $\mathcal{C}(G)$. Thus, $\mathcal{C}(G)$ can be computed using O(sort(n)) I/Os, given an MST of G.

Assigning contiguous vertex IDs By assigning contiguous IDs to the vertices in each i-component and storing this interval of vertex IDs with each component, testing membership of a vertex in an i-components becomes a simple matter of testing interval membership. To compute these vertex IDs, we compute an Euler tour of $\mathcal{C}(G)$, number the leaves (which represent the vertices of G) in the order they are visited by the Euler tour, and define the ID of every vertex to be the number given to its corresponding leaf in $\mathcal{C}(G)$. The same traversal of the Euler tour can also compute the interval of IDs assigned to the descendant leaves of each node $[x]_i$ in $\mathcal{C}(G)$, which are exactly the vertices in $[x]_i$. The Euler tour can be computed using O(sort(n)) I/Os because $\mathcal{C}(G)$ is easily seen to have size at most $2n$. The computation of vertex IDs is then easily implemented using list ranking, which takes another O(sort(n)) I/Os [CGG$^+$95].

Computing the cluster trees To compute the cluster trees, we arrange the vertices of $\mathcal{C}(G)$ in postorder and apply time-forward processing [CGG+95, Arg03] to process $\mathcal{C}(G)$ bottom-up. Initially, every leaf $[x]_0$ is labelled with the index k of the cluster V_k containing vertex x. For each such leaf, we add a pair $([x]_0, k)$ to a list C to indicate that $[x]_0$ belongs to $\mathcal{C}_k$. Then $[x]_0$ sends its cluster ID k to its parent. Every non-leaf node $[x]_i$ receives a set of cluster IDs from its children, possibly with duplications. We remove duplicates from this list and, for each remaining cluster ID k, add a pair $([x]_i, k)$ to C to indicate that node $[x]_i$ belongs to $\mathcal{C}_k$. Node $[x]_i$ forwards each such cluster ID k to its parent to ensure its addition to $\mathcal{C}_k$ unless V_k is an i-cluster, in which case $[x]_i$ is the root of $\mathcal{C}_k$.

The elimination of duplicates from the set of cluster IDs received by a node $[x]_i$ can be incorporated into the time-forward processing procedure by ensuring that the priority queue used to implement it returns these duplicates consecutively. This is easily done by using the cluster ID of each entry as a secondary key. To allow each node $[x]_i$ to identify the cluster IDs that represent i-clusters, we store the category of V_k with the cluster ID k assigned to each leaf $[x]_0$ in $\mathcal{C}_k$ and pass this information along with the cluster IDs as they are sent from node to node in $\mathcal{C}(G)$.

Once the time-forward processing step has constructed the list C, we finish the construction of the cluster trees by sorting the entries in C by their associated cluster IDs. Moreover, we sort the nodes in each cluster tree in preorder, that is, primarily by increasing lower endpoints of their associated vertex intervals and secondarily by decreasing upper endpoints of these intervals.

Lemma 7 *The total size of all cluster trees $\mathcal{C}_1, \mathcal{C}_2, \ldots, \mathcal{C}_q$ is $\mathrm{O}(n)$.*

Proof 7 *Since there are $q = \mathrm{O}(n)$ cluster trees and the size of $\mathcal{C}(G)$ is at most $2n$, it suffices to prove that every non-root node of a cluster tree $\mathcal{C}_k$ belongs to no other cluster tree. So assume there is a non-root node $[x]_i$ of a cluster tree $\mathcal{C}_k$ that also belongs to another cluster tree $\mathcal{C}_{k'}$. Then the categories of the clusters V_k and $V_{k'}$ are $j > i$ and $j' \geq i$, respectively, and $[x]_i$ contributes vertices to both clusters. This contradicts property (PP3) of a proper cluster partition.*

Corollary 2 *The cost of computing the cluster trees is $\mathrm{O}(\mathrm{sort}(n))$ I/Os.*

Proof 8 *We have already argued that computing the component tree of G from an MST of G and assigning contiguous vertex IDs to the vertices in each i-component take $\mathrm{O}(\mathrm{sort}(n))$ I/Os. The final sorting of the list C also takes $\mathrm{O}(\mathrm{sort}(n))$ I/Os because every entry in C represents a cluster tree node and, by Lemma 7, there are $\mathrm{O}(n)$ of them. To extract the cluster trees, we number the vertices of $\mathcal{C}(G)$ in postorder by applying list ranking to the Euler tour of $\mathcal{C}(G)$ [CGG+95] and then sort them by their numbers. This takes $\mathrm{O}(\mathrm{sort}(n))$ I/Os. Every message sent along an edge of $\mathcal{C}(G)$ in the time-forward processing step corresponds to a cluster tree edge. Thus, the total amount of data sent along the edges of $\mathcal{C}(G)$ is $\mathrm{O}(n)$, and the time-forward processing step takes $\mathrm{O}(\mathrm{sort}(n))$ I/Os [Arg03].*

4.3 The Shortest Path Algorithm

As already said, our shortest path algorithm is a variant of Dijkstra's algorithm: We maintain a priority queue of vertices with their tentative distances as priorities. This priority queue is implemented using the bucket heap from Section 3. At the beginning of the algorithm, we perform an UPDATE$(s, 0)$ operation on the priority queue to insert the source vertex s. Then we perform a DELETEMIN operation to retrieve a set S of vertices satisfying properties (PQ1)–(PQ3), declare their priorities to be their final distances from s, and relax their incident edges. We repeat this process until the priority queue is empty.

The key to an I/O-efficient implementation of this strategy is a method to access the adjacency lists of visited vertices I/O-efficiently. In our algorithm, we use a hierarchy of hot pools extending the hot pool used in Mehlhorn and Meyer's BFS algorithm, and we make the manipulation of these hot pools and the relaxation of edges the responsibility of DELETEMIN operations: For every vertex $x \in S$, we perform a VISIT$(x, p(x))$ operation on the priority queue, which instructs it to relax the edges incident to x. To avoid random accesses to adjacency lists, DELETEMIN operations do not relax edges immediately but perform relaxations in a batched fashion, just in time to ensure the correctness of the algorithm.

4.3.1 Hot Pools and Tree Buffers

The hot pool hierarchy consists of r hot pools $H_1, H_2, \ldots, H_r$. These hot pools store two types of edges, sorted primarily by their tails and secondarily by their heads. For each *regular edge* xy in a hot pool, the vertex cluster containing x contains an already visited vertex. We say we visit a vertex z when we insert a VISIT(z, p) operation into Q_0. Each *reverse relaxed edge* xy^R corresponds to an edge yx that has already been relaxed. We use the latter to avoid relaxing edges whose heads have already been visited.

For all $1 \leq i \leq r$, a DELETEMIN operation scans the hot pool H_i whenever it scans bucket Q_i. Thus, by Lemma 5, the frequency of inspecting hot pool H_i decreases exponentially with i. To minimize the number of times each edge is scanned, while at the same time relaxing it in time to ensure the correctness of the algorithm, we store edges in infrequently scanned hot pools and move them to more frequently scanned hot pools as their relaxation approaches. More precisely, we store a category-j edge xy in the highest hot pool H_i such that $i = j$ or no vertex in $[x]_{i-1}$ has been or will be visited before the next inspection of H_i.

To choose this hot pool H_i, for each edge xy, we use the cluster trees. We store the nodes of these cluster trees in *tree buffers* $R_1, R_2, \ldots, R_r$ associated with the hot pools. Each node $[x]_h$ in a tree buffer R_j has a label $\text{lbl}([x]_h) \geq h$. A label $\text{lbl}([x]_h) = i$ indicates that the i-component $[x]_i$ contains a vertex that has already been visited or will be visited before the next inspection of H_{i+1}.

To implement the different priority queue operations, we follow the same strategy as in Section 3: UPDATE, DELETE, and VISIT operations simply append

themselves to the end of bucket Q_0, while DeleteMin operations do the real work. Next we discuss the modifications we make to the two phases of each DeleteMin operation, in order to maintain hot pools and tree buffers, and to handle Visit operations.

Upward phase The sorting of bucket Q_0 at the beginning of the upward phase remains unchanged, as does the propagation of operations from Q_0 to Q_1. During the propagation of operations from bucket Q_i to bucket Q_{i+1}, for $1 \leq i \leq r$, we first process Delete operations as described in Section 3. Then we process Visit(x, p) and Update(x, p) operations as follows.

First we load all edge clusters containing edges that need to be relaxed. We construct a list X of all category-i vertices x such that Q_i contains a Visit(x, p) operation. Next we identify all vertices $x \in X$ such that H_i does not contain any regular edge with tail x. We generate the list of indices of the vertex clusters containing these vertices, remove duplicates from this list, retrieve the corresponding edge clusters, sort the edges they contain primarily by their tails and secondarily by their heads, and finally merge the resulting sorted edge list into H_i. We also load the corresponding cluster trees into R_i. For every edge cluster E_k merged into H_i, we merge the nodes of $\mathcal{C}_k$ into R_i so that the nodes in R_i remain sorted by increasing lower boundaries and decreasing upper boundaries of their vertex intervals. Assuming V_k is a j-cluster, we set the label of each node of $\mathcal{C}_k$ to j. The root of $\mathcal{C}_k$ may already be in R_i, as part of a different cluster tree. In this case, both copies have label j, we keep only one copy in R_i, discarding the other copy.

Next we relax category-i edges in H_i incident to visited vertices. For every Visit(x, p) operation in Q_i, we inspect all regular category-i edges and reverse relaxed edges with tail x in H_i. For each regular category-i edge xy, we add an Update$(y, p + \ell(xy))$ operation to a list U, marking this operation as being caused by edge xy, unless H_i contains the reverse relaxed edge xy^R; if it does, we ignore edge xy because vertex y has already been visited. Once this is done, we remove all regular category-i edges and reverse relaxed edges with tail x from H_i. Now we sort the Update operations in U primarily by the vertices they affect and secondarily by the tails of the edges that caused these operations and merge them into Q_i as follows. We ignore an Update(x, p) operation in U if Q_i contains a Visit(x, p') operation because the Update operation would re-insert a visited vertex into the bucket heap. If Q_i contains no Visit(x, p') operation, we insert the Update(x, p) operation into Q_i, followed by an application of rule (RR3) to merge multiple Update operations affecting x, and we add a reverse relaxed edge xy^R to a list R, where edge yx is the edge that caused the Update(x, p) operation.

Next we move edges whose relaxation is not imminent from H_i to H_{i+1}. We update the labels of the cluster tree nodes in R_i to reflect the contents of Q_i: For every node $[x]_h \in R_i$ such that Q_i contains a Visit(y, p) operation with $y \in [x]_h$, we update lbl$([x]_h)$ to h. For every node $[x]_h \in R_i$ such that Q_i does not contain a Visit(y, p) operation but contains an Update(y, p) operation with $y \in [x]_h$,

we update $\text{lbl}([x]_h)$ to $\min(\text{lbl}([x]_h), \max(h, i))$. Then we replace the label of every node $[x]_h$ with the minimum label assigned to any of its ancestors in R_i, including $[x]_h$ itself. Now we move an edge xy from H_i to H_{i+1} if $[x]_0 \in R_i$ and $\text{lbl}([x]_0) > i$ or the category of edge xy is greater than i. We move a node $[x]_h$ in R_i to R_{i+1} if $\text{lbl}([x]_h) > i$. If R_{i+1} already contains a copy of $[x]_h$, $[x]_h$ is a cluster tree root and both copies have label h. In this case, we keep only one copy of $[x]_h$ in R_{i+1}.

Next we move UPDATE operations from Q_i to Q_{i+1} as described in Section 3.3 and move all VISIT operations in Q_i to Q_{i+1} if $i < r$. If $i = r$, each VISIT operation in Q_i has been completely processed, and we discard it.

The final step is to merge the reverse relaxed edges in R into H_i. They are already in the right order because they were added to R in the same order as the corresponding UPDATE operations in U.

Downward phase The processing of priority queue buckets during the downward phase remains unaltered. In addition, when moving operations from bucket Q_j to bucket Q_{j-1}, for $1 < j \leq r$, we also move edges from H_j to H_{j-1} as necessary: After moving UPDATE operations from Q_j to Q_{j-1}, we replace the label of every node $[x]_h$ in R_j such that Q_{j-1} contains an UPDATE operation affecting a vertex in $[x]_h$ with $\min(\text{lbl}([x]_h), \max(h, j-1))$. Then we replace the label of every node in R_j with the minimum label of all its ancestors in R_j, including itself.

Now, if $j > 1$, we move a node in R_j to R_{j-1} if its label is less than j. Since cluster tree roots are easily seen to move only upwards, this cannot place duplicate nodes into R_{j-1}. We move an edge xy in H_j to H_{j-1} if the category of edge xy is less than j and $[x]_0$ is now stored in R_{j-1}. Finally, we remove every node $[x]_j$ from R_j such that R_j contains no proper descendants of $[x]_j$. This is easily checked because these descendants would be stored immediately after $[x]_j$ in R_j.

If $j = 1$, R_j contains only 0- and 1-components of G. We remove every 0-component with label 0 from R_j. After removing 0-components in this manner, we remove every 1-component that has no proper descendants in R_j. Since every edge is in category 1 or greater, there is no need to move edges to H_{j-1} in this case.

By the order in which the elements of priority queue buckets, hot pools, and tree buffers are stored, all operations performed in these two phases of a DELETEMIN operation can be implemented by scanning the affected sequences. The only exceptions are the initial sorting of bucket Q_0, the sorting of edges and cluster tree nodes when loading new edge clusters, and the updating of the labels of cluster tree nodes. Next we discuss how to do the latter I/O-efficiently. In Sections 4.3.3 and 4.3.4, we prove the correctness of our algorithm and analyze its I/O complexity.

4.3.2 Labelling and Moving Cluster Tree Nodes and Edges

In each of the two phases, the labels of cluster tree nodes in a tree buffer R_i are updated based on the contents of a bucket Q_j, where $j = i - 1$ or $j = i$. First the label of each node $[x]_h \in R_i$ is replaced with h if Q_j contains a VISIT(y, p) operation with $[x]_h = [y]_h$, or with $\min(\text{lbl}([x]_h), \max(h, j))$ if Q_j contains no such VISIT operation but an UPDATE(y, p) operation with $[x]_h = [y]_h$. Then the label of every node $[x]_h$ in R_i is replaced with the minimum label of all its ancestors in R_i.

We implement the first step by scanning R_i and Q_j twice, with the scan of R_i driving the scan of Q_j. The first time we scan R_i and Q_j, we deal with VISIT operations. For every node $[x]_h$ in R_i, we continue the scan of Q_j from the current position until we find the first VISIT(y, p) operation with $y \geq a$, where $[a, b]$ is the interval of vertices in $[x]_h$. If $y \leq b$, then $[x]_h = [y]_h$ and we set $\text{lbl}([x]_h) = h$. If $y > b$, we leave $\text{lbl}([x]_h)$ unchanged. The second scan deals with UPDATE operations in Q_j in the same manner, except that $\text{lbl}([x]_h)$ is updated to $\min(\text{lbl}([x]_h), \max(x, j))$, not h, if the scan finds an UPDATE(y, p) operation with $[x]_h = [y]_h$.

To implement the second step, we scan R_i a third time. We maintain a stack S containing all nodes in R_i that are ancestors of the current node, sorted top-down. Initially, S is empty. When processing a node $[x]_h$, we pop nodes from S until either S is empty or the topmost entry on S is an ancestor $[x]_j$ of $[x]_h$. If S is empty, we leave the label of $[x]_h$ unchanged; otherwise we replace $\text{lbl}([x]_h)$ with $\min(\text{lbl}([x]_h), \text{lbl}([x]_j))$. Once this is done, we push node $[x]_h$ onto S.

The correctness of this procedure follows immediately from the ordering of the elements in R_i and Q_j. Since we scan Q_j twice and R_i three times, and each element of R_i gives rise to two stack operations, we have the following lemma.

Lemma 8 *Updating the labels of cluster tree nodes in a tree buffer R_i based on* VISIT *and* UPDATE *operations in a bucket Q_j takes $O((|R_i| + |Q_j|)/B)$ I/Os.*

4.3.3 Correctness

Lemmas 11 and 13 below show that every vertex is visited exactly once by our algorithm and that its priority at that time equals its distance from s. To prove these lemmas, we require some lemmas that characterize where edges and cluster tree nodes are stored in the hierarchy of hot pools and tree buffers.

Lemma 9 *Let $[x]_h$ and $[x]_i$ be two nodes of the same cluster tree, and let $[x]_i$ be an ancestor of $[x]_h$. If $[x]_h \in R_{h'}$ and $[x]_i \in R_{i'}$, then $h' \leq i'$. Moreover, $[x]_i$ is removed from R_i only when $[x]_h$ has been removed or is stored in a tree buffer $R_{h'}$ with $h' < i$. Subsequently, $[x]_h$ is stored only in tree buffers $R_{h''}$ with $h'' < i$.*

Proof 9 *First we prove that $[x]_h$ cannot be stored in a higher tree buffer than $[x]_i$. Initially, nodes $[x]_h$ and $[x]_i$ are loaded into the same tree buffer. So assume*

nodes $[x]_h$ *and* $[x]_i$ *are currently in tree buffers* $R_{h'}$ *and* $R_{i'}$ *with* $h' \leq i'$. *The only changes that can invalidate the claim are moving* $[x]_h$ *from* $R_{h'}$ *to* $R_{h'+1}$ *or moving* $[x]_i$ *from* $R_{i'}$ *to* $R_{i'-1}$, *and only if* $h' = i'$. *In this case, however, we have* $\mathrm{lbl}([x]_h) \leq \mathrm{lbl}([x]_i)$. *Thus, when moving* $[x]_h$ *from* $R_{h'}$ *to* $R_{h'+1}$, *we have* $\mathrm{lbl}([x]_i) \geq \mathrm{lbl}([x]_h) > h'$, *and* $[x]_i$ *is also moved from* $R_{h'}$ *to* $R_{h'+1}$. *Similarly, when moving* $[x]_i$ *from* $R_{i'}$ *to* $R_{i'-1}$, *we have* $\mathrm{lbl}([x]_h) \leq \mathrm{lbl}([x]_i) < i'$ *and, hence,* $[x]_h$ *is also moved to* $R_{i'-1}$.

Now consider the removal of $[x]_i$ *from* R_i *by a* DeleteMin *operation. If* $[x]_h$ *is still stored in a tree buffer, then, by the first part of the lemma, it is stored in one of the tree buffers* $R_1, R_2, \ldots, R_i$. *Moreover, since* $[x]_h$ *is a proper descendant of* $[x]_i$ *and we remove* $[x]_i$ *from* R_i *only if it has no proper descendants in* R_i, $[x]_h$ *cannot be stored in* R_i. *Since the* DeleteMin *operation removes* $[x]_i$ *from* R_i, *it inspects the tree buffers* $R_1, R_2, \ldots, R_i$. *Thus,* $[x]_h$ *being stored in a tree buffer* $R_{h'}$ *with* $h' < i$ *implies that* $\mathrm{lbl}([x]_h) = h' < i$. *Since labels never increase, this implies that* $[x]_h$ *will never be moved to a tree buffer higher than* $R_{h'}$ *again.*

The next lemma proves essentially that a cluster tree node $[x]_h$ is stored in a tree buffer R_i with $i > h$ only if we can guarantee that no vertex in $[x]_h$ is visited before the next inspection of level i. By applying this lemma to $[x]_0$, we can then prove that every category-i edge reaches H_i in time for its relaxation.

Lemma 10 *When a cluster tree node* $[x]_h$ *is stored in a tree buffer* R_i *with* $i > h$, *buckets* $Q_0, Q_1, \ldots, Q_{i-1}$ *contain no* Visit(y, p) *or* Update(y, p) *operation with* $y \in [x]_{i-1}$.

Proof 10 *First assume* $[x]_h$ *is the root of a cluster tree* $\mathcal{C}_k$. *Then* $\mathrm{lbl}([x]_h) = h$ *at all times. Thus, if* $\mathcal{C}_k$ *is loaded into a tree buffer* R_i *with* $i \leq h$, $[x]_h$ *is never moved to a tree buffer higher than* R_h. *If* $\mathcal{C}_k$ *is loaded into a tree buffer* R_i *with* $i > h$, *then buckets* $Q_0, Q_1, \ldots, Q_{i-1}$ *are empty at this time, and the lemma holds. The downward phase of the same* DeleteMin *operation moves* $[x]_h$ *to* R_h, *and (this copy of)* $[x]_h$ *will not be stored in a tree buffer higher than* R_h *again.*

So assume $[x]_h$ *is not a cluster tree root. Initially,* $[x]_h$ *is loaded into a tree buffer* R_i *as part of the cluster tree* $\mathcal{C}_k$ *containing* $[x]_h$. *At this time, buckets* $Q_0, Q_1, \ldots, Q_{i-1}$ *are empty. Hence, the lemma holds. Subsequently, the lemma may be invalidated by moving node* $[x]_h$ *from a tree buffer* R_i *to* R_{i+1} *or by* Visit *or* Update *operations inserted into* $Q_0, Q_1, \ldots, Q_{i-1}$ *while* $[x]_h \in R_i$.

First assume moving $[x]_h$ *from* R_i *to* R_{i+1} *invalidates the lemma. Since buckets* $Q_0, Q_1, \ldots, Q_{i-1}$ *are empty at this time, this implies that there is a* Visit(y, p) *or* Update(y, p) *operation in* Q_i *with* $y \in [x]_i$ *and which is not moved to* Q_{i+1} *when moving* $[x]_h$ *to* R_{i+1}. *Since every* Visit *operation in* Q_i *is moved to* Q_{i+1} *when inspecting level* i, *the operation must be an* Update(y, p) *operation with* $p < q_{i+1}$. *Let* $[x]_{i'}$ *be the minimal* i'*-component such that* $[x]_{i'} = [x]_i$. *If* $[x]_{i'}$ *is an ancestor of the root of* $\mathcal{C}_k$, *then* $\mathrm{lbl}([x]_h) \leq i' \leq i$, *contradicting that we move* $[x]_h$ *to* R_{i+1}. *So* $[x]_{i'}$ *belongs to* $\mathcal{C}_k$. *Since* $[x]_h \in R_i$, *Lemma 9*

implies that $[x]_{i'}$ is not stored in a tree buffer below R_i. Since the lemma held before moving $[x]_h$ to R_{i+1}, the UPDATE(y,p) *operation in Q_i implies that $[x]_{i'}$ is stored in a tree buffer no higher than R_i. Thus, $[x]_{i'} \in R_i$. This in turn implies that $\mathrm{lbl}([x]_h) \leq \mathrm{lbl}([x]_{i'}) \leq i$ by the time we move $[x]_h$ to R_{i+1}. Once again, this is a contradiction.*

So consider updates to the contents of buckets $Q_0, Q_1, \ldots, Q_{i-1}$ that happen while $[x]_h$ is stored in a tree buffer R_i. There are two types of such updates: UPDATE *operations may be moved from Q_i to Q_{i-1} during a downward phase of a* DELETEMIN *operation, or new* VISIT *or* UPDATE *operations may be inserted into buckets $Q_0, Q_1, \ldots, Q_{i-1}$ during an upward phase.*

If the movement of an UPDATE(y,p) *operation from Q_i to Q_{i-1} invalidates the lemma, then $y \in [x]_{i-1}$ and the label of $[x]_h$ is at least i because otherwise $[x]_h$ would be moved to R_{i-1}. Using the same argument that shows that moving $[x]_h$ from R_i to R_{i+1} cannot invalidate the lemma, this leads to a contradiction.*

If we insert an UPDATE(y,p) *operation with $y \in [x]_{i-1}$ into a bucket Q_j with $j < i$, Q_j contains a* VISIT(z,p') *operation, where zy is a category-j edge. This implies that $z \in [x]_{i-1}$, a contradiction. Similarly, a* VISITI(y,p) *operation inserted into Q_0 replaces an* UPDATE(y,p) *operation in Q_0, contradicting that the lemma held before inserting the* VISIT(y,p) *operation into Q_0.*

Corollary 3 *When a* VISIT(x,p) *operation reaches bucket Q_i, every category-i edge incident to x that has not been relaxed yet is stored in H_i unless the edge cluster containing these edges has not been loaded yet.*

Proof 11 *Consider a category-i edge xy that has not been relaxed yet and assume the edge cluster E_k containing it is loaded into a hot pool before the* VISIT(x,p) *operation reaches bucket Q_i. If E_k is loaded after vertex x is visited, it is loaded into a hot pool H_h with $h < i$ because the* VISIT(x,p) *operation reaches Q_i during the first inspection of level i after visiting vertex x. Thus, since xy is a category-i edge, it reaches H_i at the same time the* VISIT(x,p) *operation reaches Q_i.*

If E_k is loaded into a hot pool before x is visited, observe that, by Lemma 10, the cluster tree node $[x]_0$ cannot be stored in any tree buffer when x is visited. However, since xy is a category-i edge, it can be stored in a hot pool H_h with $h > i$ only if node $[x]_0$ is stored in the corresponding tree buffer R_h. Thus, since edge xy has not been relaxed yet, it is stored in a hot pool H_h with $h \leq i$ at the time vertex x is visited. The same argument as in the previous case now shows that edge xy reaches H_i no later than the VISIT(x,p) *operation reaches Q_i.*

We are now ready to prove that our algorithm visits every vertex exactly once.

Lemma 11 *Every vertex x is visited exactly once.*

Proof 12 *It is easily verified that every vertex is visited at least once, assuming the graph is connected. So assume there is a vertex that is visited twice, and let x be the first such vertex. For this to happen, we have to insert an* UPDATE(x,p)

operation into a bucket Q_i after x is visited for the first time. This insertion is the result of relaxing a category-i edge yx.

When the relaxation of edge yx inserts the UPDATE(x, p) *operation into Q_i, Q_i contains a* VISIT(y, p') *operation. The* VISIT(x, p'') *operation inserted into Q_0 the first time vertex x is visited reaches Q_i no later than this* VISIT(y, p') *operation.*

If the VISIT(x, p'') *and* VISIT(y, p') *operations reach bucket Q_i at the same time, the relaxation of edge yx does not insert an* UPDATE$(x, p' + \ell(xy))$ *operation into Q_i.*

If the VISIT(x, p'') *operation reaches Q_i before the* VISIT(y, p') *operation, observe that, by the choice of x, the* VISIT(y, p') *operation is the first* VISIT *operation affecting y that reaches Q_i. Thus, when the* VISIT(x, p'') *operation reaches Q_i, there is no* VISIT *operation affecting y in Q_i, and the relaxation of edge xy leads to the insertion of the reverse relaxed edge yx^R into H_i. When the* VISIT(y, p') *operation reaches Q_i, it finds edge yx^R in H_i and, hence, edge yx is not relaxed. Therefore, again no* UPDATE$(x, p' + \ell(yx))$ *operation is inserted into Q_i.*

Lemma 11 and Corollary 3 immediately imply the following corollary.

Corollary 4 *Every edge cluster is loaded once.*

The final lemma we need in order to prove the correctness of our algorithm shows that the bucket heap continues to operate as a correct priority queue even though we insert UPDATE operations into buckets Q_i with $i > 0$ as a result of edge relaxations instead of inserting these operations into Q_0. The lemma shows that each UPDATE(y, p') operation inserted into a bucket Q_i is the last operation in S_y and satisfies $q_i \leq p' \leq p_{\max} + L$. Since $q_i \geq p_{\max}$, for $i \geq 1$, this implies in particular that the updates we perform satisfy the bounded update condition. Thus, neither the bounded update condition nor Lemma 1 are violated, and the remainder of the correctness proof of the bucket heap continues to hold.

Lemma 12 *Every* UPDATE(y, p') *operation inserted into Q_i as a result of processing a* VISIT(x, p) *operation in Q_i satisfies $q_i \leq p' \leq p_{max} + L$. It is the last operation in S_y and, thus, is correctly recorded as an update of the priority of y.*

Proof 13 *The second claim is easily seen to be true because, at the time the* VISIT(x, p) *operation in Q_i is processed, buckets $Q_0, Q_1, \ldots, Q_{i-1}$ are empty and bucket Q_i contains no* DELETE *operations.*

To prove the first claim, observe that processing a VISIT(x, p) *operation in Q_i inserts an* UPDATE(y, p') *operation into Q_i only if xy is a category-i edge. At the time the* VISIT(x, p) *operation was inserted into Q_0, it satisfied $q_0 \leq p \leq p_{max}$. Since $\ell(xy) \leq L$ and p_{max} does not decrease (see page 6), this implies that $p' = p + \ell(xy) \leq p_{max} + L$. Level i is not inspected between the insertion of the* VISIT(x, p) *operation into Q_0 and the inspection of level i that relaxes edge xy and inserts the* UPDATE(y, p') *operation into Q_i. Thus, the current*

value of q_i and the value of q_0 at the time x is visited satisfy $q_i \leq q_0 + 2^{i-1}$, by Lemma 2. Since xy is a category-i edge, we have $\ell(xy) \geq 2^{i-1}$, that is, $p' = p + \ell(xy) \geq q_0 + 2^{i-1} \geq q_i$.

We can now prove the correctness of our algorithm.

Lemma 13 *Every vertex x is visited with priority $p(x) = \text{dist}(s, x)$.*

Proof 14 *Since the priorities of vertices are updated only through edge relaxations, we have $p(x) \geq \text{dist}(s, x)$ at all times, for all x. So assume there exists a vertex x that is visited with a priority $p(x) > \text{dist}(s, x)$, that x is the vertex with minimum distance from s that is visited with a priority greater than its distance, and that the category of the last edge yx on the shortest path $\pi(s, x)$ from s to x is i. Then the predecessor y of x on $\pi(s, x)$ is visited with priority $p(y) = \text{dist}(s, y)$. Since $\text{dist}(s, y) = \text{dist}(s, x) - \ell(yx) \leq \text{dist}(s, x) - 1 < p(x) - 1$, property (PQ2) of the bucket heap implies that the* DeleteMin *operation visiting x must succeed the one visiting y. The* Visit$(y, \text{dist}(s, y))$ *operation inserted into Q_0 when vertex y is visited reaches bucket Q_i during the next inspection of level i and inserts an* Update$(x, \text{dist}(s, y) + \ell(yx))$ *operation into Q_i unless vertex x is visited before this inspection. By Lemma 12, this updates x's priority to $\text{dist}(s, y) + \ell(yx) = \text{dist}(s, x)$. Thus, it suffices to prove that level i is inspected before vertex x is visited.*

Assume the contrary. At the time y is visited, we have $p(y) = \text{dist}(s, y) \geq q_0$ and, hence, by Lemma 2, $q_i \leq \text{dist}(s, y) + 2^{i-1}$. Since q_i is changed only when inspecting a level $j \geq i$, we also have $q_i \leq \text{dist}(s, y) + 2^{i-1}$ at the time x is visited. Since we assume $p(x) > \text{dist}(s, x) = \text{dist}(s, y) + \ell(yx)$, every Update$(x, p)$ *operation in the priority queue at this time satisfies $p \geq p(x) > \text{dist}(s, y) + 2^{i-1} \geq q_i$. Therefore, x cannot be visited before the next inspection of level i.*

4.3.4 I/O Complexity

We divide the cost of the algorithm into the costs of manipulating the various data structures.

Lemma 14 *The cost of manipulating priority queue buckets is $\text{O}(\text{sort}(m) + (m/B) \log L)$ I/Os.*

Proof 15 *Since every vertex is visited exactly once (Lemma 11), we insert n* Visit *and n* Delete *operations into the priority queue. Every* Update *operation inserted into the priority queue is the result of an edge relaxation, and there are $2m$ such relaxation, one per edge and endpoint. Thus, we process $\text{O}(m)$ operations. The lemma now follows from Lemma 4 after observing that the coupling of the priority queue with the hot pool hierarchy increases the cost of inspecting each level of the bucket heap by only a constant factor and, just as* Update *and* Delete *operations, each* Visit *operation remains in a given bucket Q_i for $\text{O}(1)$ inspections of level i.*

To analyze the cost of manipulating tree buffers, we bound how long each cluster tree node remains in a tree buffer. We consider root and non-root nodes separately.

Lemma 15 *Let $[x]_h$ be a cluster tree node stored in a tree buffer R_i and that is not the root of its cluster tree. When level i is inspected, node $[x]_h$ is moved to R_{i+1} or all vertices in $[x]_h$ are visited before level i is inspected $6\mu + 8$ more times.*

Proof 16 *If the inspection of level i does not move $[x]_h$ to R_{i+1}, then $\mathrm{lbl}([x]_h) \leq i$. This implies that either V_k is a j-cluster with $j \leq i$, where $\mathcal{C}_k$ is the cluster tree containing $[x]_h$, or there exists a vertex $y \in [x]_i$ such that one of the buckets $Q_0, Q_1, \ldots, Q_i$ contains or contained an $\textsc{Update}(y, p)$ operation with $p < q_{i+1}$. (This is true even if $\mathrm{lbl}([x]_h) \leq i$ due to a $\textsc{Visit}(y, p)$ operation with $y \in [x]_h$ because, before visiting y, Q_0 contained the corresponding $\textsc{Update}(y, p)$ operation.)*

First assume V_k is a j-cluster with $j \leq i$. The loading of $\mathcal{C}_k$ into a tree buffer is caused by a $\textsc{Visit}(y, \mathrm{dist}(s, y))$ operation, for some vertex $y \in V_k$. The DeleteMin *operation o_1 inserting this operation into Q_0 visits only vertices with priority at least $\mathrm{dist}(s, y) - 1$, by property (PQ2) of the bucket heap. Since $[x]_h \in \mathcal{C}_k$, there exists a vertex $z' \in [x]_h$ that belongs to V_k. By property (C2) of a j-cluster, this implies that $\mathrm{dist}(y, z) \leq \mu 2^j \leq \mu 2^i$, that is, $\mathrm{dist}(s, z) \leq \mathrm{dist}(s, y) + \mu 2^i$, for all $z \in [x]_h$. Now let o_2 be the last* DeleteMin *operation visiting a vertex $z \in [x]_h$. Since $\mathrm{dist}(s, z) \leq \mathrm{dist}(s, y) + \mu 2^i$, Lemma 5 shows that at most $2 + 3(\mu 2^i + 1)/2^{i-1} \leq 6\mu + 8$* DeleteMin *operations between o_1 and o_2, inclusive, inspect level i.*

Now assume V_k is a j-cluster with $j > i$. Then $\mathrm{lbl}([x]_h) \leq i$ implies that one of the buckets $Q_0, Q_1, \ldots, Q_i$ contains or contained an $\textsc{Update}(y, p)$ operation with $y \in [x]_i$ and satisfying $\mathrm{dist}(s, y) \leq p < q_{i+1} \leq q_0 + 2^i$ with respect to the values of q_0 and q_{i+1} at the time this operation was in this buckets. Since the value of q_0 does not decrease, we also have $\mathrm{dist}(s, y) < q_0 + 2^i$ for any subsequent value of q_0. Now let $[x]_{i'}$ be the minimal i'-component such that $[x]_{i'} = [x]_i$. Then $[x]_{i'} \in \mathcal{C}_k$, that is, $[x]_{i'} \cap V_k \neq \emptyset$. By property (C3) of a j-cluster and because $i' \leq i < j$, this implies that the diameter of $[x]_{i'}$ is at most $\mu 2^{i'} \leq \mu 2^i$. Hence, every vertex $z \in [x]_h \subseteq [x]_{i'}$ satisfies $\mathrm{dist}(s, z) \leq \mathrm{dist}(s, y) + \mu 2^i$. Now let o_1 be the current DeleteMin *operation inspecting level i. This operation visits no vertex with priority less than $q_0 > \mathrm{dist}(s, y) - 2^i$, while the last* DeleteMin *operation o_2 visiting a vertex $z \in [x]_h$ visits a vertex with priority at most $\mathrm{dist}(s, y) + \mu 2^i$. By Lemma 5, this implies that at most $2 + 3(\mu + 1)2^i/2^{i-1} = 6\mu + 8$ operations between o_1 and o_2, inclusive, inspect level i.*

Corollary 5 *Every non-root cluster tree node $[x]_h$ contributes $\mathrm{O}(\log L/B + \mu(r - h + 1)/B)$ I/Os to the scanning cost of tree buffers.*

Proof 17 *A tree buffer R_i is accessed when inspecting level $i - 1$ or i. By Observation 1, at least every other inspection of level $i - 1$ is followed by an inspection of level i. Hence, the number of accesses to R_i while $[x]_h \in R_i$ is*

within a constant factor of the number of times level i is inspected during this time. Since there are $r = \mathrm{O}(\log L)$ tree buffers, it therefore suffices to show that level i is inspected $\mathrm{O}(1)$ times while $[x]_h \in R_i$ if $1 \leq i < h$, and $\mathrm{O}(\mu)$ times if $h \leq i \leq r$.

For $i < h$, if $[x]_h \in R_i$, inspecting level i moves $[x]_h$ to R_{i+1} because $\mathrm{lbl}([x]_h) \geq h$ at all times. For the same reason $[x]_h$ is not moved back to R_i at a later time. Thus, level i is inspected once while $[x]_h \in R_i$.

For $i \geq h$, if the inspection of level i does not move $[x]_h$ to R_{i+1}, then, by Lemma 15, all vertices in $[x]_h$ are visited before level i is inspected $6\mu + 8$ more times. Thus, it suffices to prove that $[x]_h$ is removed from R_i before the last vertex in $[x]_h$ is visited. We treat the cases $i > h$ and $i = h$ separately.

For $i > h$, it suffices to prove that the last vertex $z \in [x]_h$ is visited after the cluster tree $\mathcal{C}_k$ containing $[x]_h$ is loaded into a hot pool. Indeed, by Lemma 10, $[x]_h \notin R_i$ when z is visited. Thus, $[x]_h$ is removed from R_i before z is visited.

Since $[x]_h \in \mathcal{C}_k$, there exists a vertex $y \in V_k$ with $y \in [x]_h$. If z is visited before $\mathcal{C}_k$ is loaded into a tree buffer, $\mathcal{C}_k$ is loaded after y is visited. Since $y \in [x]_h$ and $[x]_h$ is composed of at least two $(h-1)$-components (it is a minimal h-component), y has an incident category-j edge with $j \leq h$. Thus, $\mathcal{C}_k$ is loaded no later than the first inspection of level h after visiting y. This implies that, when $\mathcal{C}_k$ is loaded into a tree buffer R_j with $j \leq h$, the corresponding bucket Q_j contains a $\textsc{Visit}(y, p)$ *operation. Since $y \in [x]_h$, this would result in $\mathrm{lbl}([x]_h) = h$, and $[x]_h$ would never be stored in a tree buffer higher than R_h, a contradition.*

For $i = h$, note that the last vertex of any proper descendant $[x]_{h'}$ of $[x]_h$ in $\mathcal{C}(G)$ is visited no later than the last vertex in $[x]_h$. Thus, the argument for the case $i > h$ shows that $[x]_{h'} \notin R_h$ when we visit the last vertex in $[x]_h$. Since we remove $[x]_h$ from R_h once R_h contains no proper descendants of $[x]_h$, this shows that $[x]_h$ is removed from R_h before the last vertex in $[x]_h$ is visited.

By Corollary 5 and because there are only $\mathrm{O}(n)$ cluster tree nodes (Lemma 7), the total cost of scanning the non-root nodes of all cluster trees is $\mathrm{O}((\mu n \log L)/B) = \mathrm{O}((\mu m \log L)/B)$ I/Os. Next we bound the cost of scanning cluster tree roots.

Lemma 16 *The cost of scanning all cluster tree roots is $\mathrm{O}((m/B) \log L)$ I/Os.*

Proof 18 *The label of a cluster tree root $[x]_h$ equals h at all times. Thus, if $[x]_h \in R_i$ with $i < h$, the inspection of level i moves $[x]_h$ to R_{i+1}; if $i > h$, the downward phase of the* DeleteMin *operation inspecting level i moves $[x]_h$ to R_h. Thus, $[x]_h$ is scanned $\mathrm{O}(1)$ times per level $i \neq h$. Since there are $\mathrm{O}(\log L)$ tree buffers and at most n cluster tree roots, this bounds the cost of scanning all cluster tree roots as part of tree buffers R_i with $i \neq h$ by $\mathrm{O}((n/B) \log L)$ I/Os.*

Now consider the scanning cost of a cluster tree root $[x]_h \in R_h$. The first cluster tree $\mathcal{C}_k$ with root $[x]_h$ is loaded into a tree buffer in response to a $\textsc{Visit}(y, \mathrm{dist}(s, y))$ *operation with $y \in V_k \subseteq [x]_h$. The* DeleteMin *operation o_1 visiting y visits only vertices with priorities greater than $\mathrm{dist}(s, y) - 1$, by property (PQ2). If $\mathrm{diam}([x]_h)$ is the diameter of $[x]_h$, then every vertex $z \in [x]_h$ satisfies $\mathrm{dist}(s, z) \leq \mathrm{dist}(s, y) + \mathrm{diam}([x]_h)$. Thus, by Lemma 5, if*

o_2 is the last DeleteMin *operation visiting a vertex in $[x]_h$, at most $2 + 3(\mathrm{diam}([x]_h) + 1)/2^{h-1} \leq 5 + 3\mathrm{diam}([x]_h)/2^{h-1}$* DeleteMin *operations between o_1 and o_2, inclusive, inspect level h. By the arguments in the proof of Corollary 5, $[x]_h$ is part of R_h only between operations o_1 and o_2 and, thus, contributes $\mathrm{O}((1+\mathrm{diam}([x]_h)/2^h)/B)$ I/Os to the scanning cost of R_h. Now let m_i be the number of category-i edges in G, and let q_h be the number of h-clusters. Then the total diameter of all h-components is less than $\sum_{i=1}^{h} 2^i m_i$ and the total scanning cost of cluster tree roots $[x]_h$ in R_h is $\mathrm{O}\Big(\Big(q_h + \sum_{i=1}^{h} 2^i m_i/2^h\Big)/B\Big)$ I/Os. By summing over all h, we obtain that the total scanning cost of cluster tree roots in tree buffers corresponding to their categories is*

$$\sum_{h=1}^{r} \mathrm{O}\left(\frac{q_h + \sum_{i=1}^{h} 2^i m_i/2^h}{B}\right) = \mathrm{O}\left(\frac{q}{B}\right) + \mathrm{O}\left(\sum_{h=1}^{r}\sum_{i=1}^{h} \frac{2^i m_i}{2^h B}\right)$$
$$= \mathrm{O}\left(\frac{n}{B}\right) + \mathrm{O}\left(\sum_{i=1}^{r} \frac{m_i}{B} \sum_{h=0}^{+\infty} \frac{1}{2^h}\right) = \mathrm{O}\left(\frac{n}{B}\right) + \mathrm{O}\left(\sum_{i=1}^{r} \frac{m_i}{B}\right) = \mathrm{O}\left(\frac{m}{B}\right) \text{ I/Os.}$$

We divide the cost of manipulating hot pools into (1) loading edge clusters into hot pools, (2) scanning regular edges, and (3) scanning reverse relaxed edges.

Lemma 17 *The cost of loading edge clusters is $\mathrm{O}(n/\mu + \mathrm{sort}(m))$ I/Os.*

Proof 19 *The cost of loading edge clusters includes sorting and scanning* Visit *operations to determine the edge clusters to be loaded, retrieving the identified clusters, and sorting and scanning the retrieved edges to merge them into the current hot pool. Sorting and scanning* Visit *operations costs $\mathrm{O}(\mathrm{sort}(n))$ I/Os because there are only n such operations (Lemma 11) and each such operation is involved in only one such sorting and scanning pass, namely while inspecting the level corresponding to the category of the visited vertex. Retrieving the identified clusters takes $\mathrm{O}(n/\mu + m/B)$ I/Os because each cluster is loaded once (Corollary 4), there are $\mathrm{O}(n/\mu)$ clusters, and their total size is $2m$. By the same argument, the cost of sorting and scanning the retrieved edges to merge them into the hot pools is $\mathrm{O}(\mathrm{sort}(m))$ I/Os. Summing these costs proves the lemma.*

Lemma 18 *A category-i edge xy contributes $\mathrm{O}(\log L/B + \mu(r-i+1)/B)$ I/Os to the scanning cost of hot pools.*

Proof 20 *By the same arguments as in the proof of Corollary 5, edge xy is scanned $\mathrm{O}(1)$ times per hot pool $H_1, H_2, \ldots, H_{i-1}$. Thus, it contributes $\mathrm{O}(\log L/B)$ I/Os to the scanning cost of these hot pools. When edge xy is stored in a hot pool H_h with $h \geq i$, it either moves to H_{h+1} the next time level h is inspected or $\mathrm{lbl}([x]_0) \leq h$. The latter implies that either x is part of a j-cluster with $j \leq h$ or the h-component $[x]_h$ contains a vertex y such that one of the buckets $Q_0, Q_1, \ldots, Q_h$ contains or contained an* Update(y, p) *operation with $p < q_{h+1}$. As shown in the proof of Lemma 15, this implies that vertex x is visited before*

level h is inspected $6\mu + 8$ more times. Thus, by Corollary 3, edge xy is relaxed before H_h is inspected $6\mu + 9$ more times. By Corollary 3, edge xy is stored in hot pool H_i immediately before it is relaxed, that is, if $h > i$, edge xy is moved from H_h to H_i before its relaxation. If $h = i$, edge xy is removed from H_i when it is relaxed. Thus, edge xy contributes $O(\mu/B)$ I/Os to the scanning cost of each hot pool H_h with $h \geq i$. By summing over $i \leq h \leq r$, we obtain the bound claimed in the lemma.

Lemma 19 *The cost of scanning reverse relaxed edges is $O(m/B)$ I/Os.*

Proof 21 *If the category of edge xy is i, the reverse relaxed edge xy^R is never stored in a hot pool other than H_i. If xy^R is ever inserted into H_i, it is inserted after vertex y is visited by a* DeleteMin *operation o_1 and is removed from H_i by the first inspection of level i after a subsequent* DeleteMin *operation o_2 visits vertex x. If p_1 and p_2 are the minimum priorities of the vertices returned by o_1 and o_2, respectively, we have $p_1 > \text{dist}(s, y) - 1$ and $p_2 \leq \text{dist}(s, x)$. Since xy is a category-i edge, we have $\ell(yx) < 2^i$ and, hence, $p_2 \leq \text{dist}(s, x) < \text{dist}(s, y) + 2^i < p_1 + 2^i + 1$. By Lemma 5, this implies that H_i is scanned $O(1)$ times while edge $xy^R \in H_i$. Since there are $2m$ reverse relaxed edges, the lemma follows.*

Lemma 20 *Manipulating the temporary lists U and R takes $O(\text{sort}(m))$ I/Os.*

Proof 22 *The cost of manipulating these lists is bounded by the cost of sorting them. The total number of elements added to these lists is $O(m)$, since every addition of an* Update *operation to U and every addition of an edge to R can be charged to one of the $2m$ edge relaxations performed by the algorithm. Thus, the cost of manipulating U and R is $O(\text{sort}(m))$ I/Os.*

To summarize, Lemma 13 establishes the correctness of our algorithm, while Theorem 3, Lemmas 14, 16, 17, 18, 19, 20, and Corollary 5 bound its cost by $O(n/\mu + (\mu m \log B)/B + \text{MST}(n, m))$ I/Os. The first two terms in this bound are balanced by choosing $\mu = \sqrt{nB/(m \log L)}$, which proves Theorem 2.

5 Conclusions

In this paper, we have presented single-source shortest path algorithms that achieve a better performance than one I/O per vertex on sparse undirected graphs with random edge lengths or edge lengths that are bounded by a parameter $L = 2^{o(B)}$. Recent extensions of the ideas in this paper include an algorithm that achieves the same on undirected graphs with arbitrary edge lengths [MZ06] and a cache-oblivious algorithm that almost matches the performance of the worst-case efficient algorithm presented here [ALZ07]. In [Cho07, Section 4], the worst-case efficient algorithm presented in this paper has been used to speed up I/O-efficient all-pairs shortest path computations on undirected graphs.

While all these results show that clustering ideas are effective to speed up shortest path computations on massive graphs—a fact that has been verified

also in practice [[ADM06]; [AMO07]; [MO09]]—the true I/O complexity of the SSSP problem on undirected graphs remains open. No better lower bound than $\Omega(\text{sort}(n+m))$ I/Os is known. An even harder challenge is to make progress on directed graphs. The clustering techniques used here and in the above papers fail on directed graphs. Even the simpler BFS and shortest path algorithms by [KS96] and [MR99] are slower on directed graphs than on undirected graphs, as they require a more costly data structure [BGVW00] to remember explored vertices in the directed case.

References

[AAY06] Pankaj K. Agarwal, Lars Arge, and Ke Yi. I/O-efficient batched union-find and its applications to terrain analysis. In *Proceedings of the 22nd ACM Symposium on Computational Geometry*, pages 167–176, 2006.

[ABT04] Lars Arge, Gerth Stølting Brodal, and Laura Toma. On external-memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53(2):186–206, 2004.

[ABW02] James Abello, Adam L. Buchsbaum, and Jeffery Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

[ADM06] Deepak Ajwani, Roman Dementiev, and Ulrich Meyer. A computational stuy of external-memory BFS algorithms. In *Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*, pages 601–610, 2006.

[ALZ07] Luca Allulli, Peter Lichodzijewski, and Norbert Zeh. A faster cache-oblivious shortest-path algorithm for undirected graphs with bounded edge lengths. In *Proceedings of the 18th ACM-SIAM Symposium on Discrete Algorithms*, pages 910–919, 2007.

[AMO07] Deepak Ajwani, Ulrich Meyer, and Vitaly Osipov. Improved external memory BFS implementation. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments*, pages 3–12, 2007.

[AMOT90] Ravindra K. Ahuja, Kurt Mehlhorn, James B. Orlin, and Robert E. Tarjan. Faster algorithms for the shortest path problem. *Journal of the ACM*, 37(2):213–223, 1990.

[AMT04] Lars Arge, Ulrich Meyer, and Laura Toma. External memory algorithms for diameter and all-pairs shortest-paths on sparse graphs. In *Proceedings of the 31st International Colloquium on Automata, Languages, and Programming*, volume 3124 of *Lecture Notes in Computer Science*, pages 146–157. Springer-Verlag, 2004.

[Arg03] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[ATZ03] Lars Arge, Laura Toma, and Norbert Zeh. I/O-efficient algorithms for planar digraphs. In *Proceedings of the 15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 85–93, 2003.

[AV88] Alok Aggarwal and Jeffrey Scott Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[Bel58] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[BFMZ04] Gerth Stølting Brodal, Rolf Fagerberg, Ulrich Meyer, and Norbert Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory*, volume 3111 of *Lecture Notes in Computer Science*, pages 480–492. Springer-Verlag, 2004.

[BGVW00] Adam L. Buchsbaum, Michael H. Goldwasser, Suresh Venkatasubramanian, and Jeffery Westbrook. On external memory graph traversal. In *Proceedings of the 11th ACM-SIAM Symposium on Discrete Algorithms*, pages 859–860, 2000.

[CGG+95] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *Proceedings of the 6th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[Cho07] Rezaul Alam Chowdhury. *Cache-Efficient Algorithm and Data Structures: Theory and Experimental Evaluation*. PhD thesis, Department of Computer Sciences, The University of Texas at Austin, 2007.

[CR04] Rezaul Alam Chowdhury and Vijaya Ramachandran. Cache-oblivious shortest paths in graphs using buffer heap. In *Proceedings of the 16th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 245–254, 2004.

[CR05] Rezaul Alam Chowdhury and Vijaya Ramachandran. External-memory exact and approximate all-pairs shortest-paths in undirected graphs. In *Proceedings of the 16th ACM-SIAM Symposium on Discrete Algorithms*, pages 735–744, 2005.

[Dij59] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[FF62] Lestor R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[FT87] Michael L. Fredman and Robert E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34(3):596–615, 1987.

[HT06] Herman J. Haverkort and Laura Toma. I/O-efficient algorithms on near-planar graphs. In *Proceedings of the 7th Latin American Symposium on Theoretical Informatics*, volume 3887 of *Lecture Notes in Computer Science*, pages 580–591. Springer-Verlag, 2006.

[KS96] Vijay Kumar and Eric J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing*, pages 169–176, 1996.

[MM02] Kurt Mehlhorn and Ulrich Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the 10th European Symposium on Algorithms*, volume 2461 of *Lecture Notes in Computer Science*, pages 723–735. Springer-Verlag, 2002.

[MO09] Ulrich Meyer and Vitaly Osipov. Design and implementation of a practical I/O-efficient shortest paths algorithm. In *Proceedings of the Workshop on Algorithm Engineering and Experiments*, pages 85–96, 2009.

[MR99] Kameshwar Munagala and Abhiram Ranade. I/O-complexity of graph algorithms. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 687–694, 1999.

[MSS03] Ulrich Meyer, Peter Sanders, and Jop Sibeyn, editors. *Algorithms for Memory Hierarchies: Advanced Lectures*, volume 2625 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[MZ01] Anil Maheshwari and Norbert Zeh. I/O-efficient algorithms for graphs of bounded treewidth. In *Proceedings of the 12th ACM-SIAM Symposium on Discrete Algorithms*, pages 89–90, 2001.

[MZ06] Ulrich Meyer and Norbert Zeh. I/O-efficient undirected shortest paths with unbounded edge lengths. In *Proceedings of the 14th European Symposium on Algorithms*, volume 4168 of *Lecture Notes in Computer Science*, pages 540–551. Springer-Verlag, 2006.

[PR02] Seth Pettie and Vijaya Ramachandran. Computing shortest paths with comparisons and additions. In *Proceedings of the 13th ACM-SIAM Symposium on Discrete Algorithms*, pages 267–276, 2002.

[Tho99] Mikkel Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *Journal of the ACM*, 46(3):362–394, 1999.

[Tho00] Mikkel Thorup. Floats, integers, and single source shortest paths. *Journal of Algorithms*, 35(2):189–201, 2000.

[VEBKZ77] P. Van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Mathematical Systems Theory*, 10:99–127, 1977.

[Vit08] Jeffrey S. Vitter. *Algorithms and Data Structures for External Memory*. now Publishers, 2008.

A A Simple Algorithm for Random Edge Lengths

In this section, we show that a simpler algorithm than in Section 4 achieves comparable performance under the assumption that the edge lengths are chosen uniformly at random. [MO09] implemented this algorithm and observed much better performance of the algorithm on real-world inputs in their experiments than predicted by the analysis in this section. The following theorem states the main result of this section.

Theorem 4 *The single-source shortest path problem on an undirected graph with n vertices, m edges, and uniformly random edge lengths in the interval $(0, 1]$ can be solved using expected $\mathrm{O}\Big(\sqrt{(nm \log B)/B} + \mathrm{MST}(n, m)\Big)$ I/Os.*

To keep the terminology consistent with the discussion in Section 4, we describe the algorithm under the assumption that the edge lengths are chosen uniformly at random from the interval $(0, B]$. Since we can scale all edge lengths by a factor of B without affecting shortest paths, this is equivalent to the assumption in Theorem 4.

There are two main differences to the algorithm in Section 4: Firstly, the algorithm in this section uses the simple clustering procedure from [MM02]'s BFS algorithm, which ignores edge lengths (see Section 2.3). Secondly, the placement of edges into hot pools is determined exclusively by edge categories: there is no need for cluster trees and tree buffers. The resulting algorithm, discussed in Section A.1, can deal with edge lengths between 1 and B, which we call *long*. In order to correctly compute shortest paths that include *short* edges of length less than 1, we need to deal with these edges specially. This is discussed in Section A.2. Essentially, if the edge lengths are distributed uniformly, there cannot be too many short edges, which allows us to deal with them in a fairly naïve manner.

A.1 An Algorithm for Long Edges

Throughout this subsection, we assume all edges are long. We start by dividing the vertex set of G into vertex clusters $V_1, V_2, \ldots, V_q$ as in [MM02]'s BFS algorithm (see Section 2.3) and store the corresponding edge clusters $E_1, E_2, \ldots, E_q$ consecutively on disk. As in the BFS algorithm, this can be done using $O(\text{MST}(n, m))$ I/Os. Next we run the shortest path algorithm from Section 4.3 with parameter $L = B$ and with the following simplification of DeleteMin operations. When the upward phase inspects a hot pool H_i, we move an edge xy from H_i to H_{i+1} if and only if its category is greater than i. Similarly, when the downward phase inspects a hot pool H_i, we move an edge xy from H_i to H_{i-1} if and only if its category is less than i. This is done without any regard for how close edge xy is to being relaxed and, thus, can be implemented without cluster trees or tree buffers. The hot pool where a category-i edge is stored is different from where the algorithm from Section 4.3 would store the edge only when the latter would store the edge in a hot pool H_j with $j \geq i$: in this case, the algorithm in this section stores the edge in a hot pool $H_{j'}$ with $i \leq j' \leq j$. Thus, the correctness of the algorithm is unaffected. The only component of the I/O complexity of the algorithm affected by potentially storing edges in more frequently scanned hot pools is the hot pool scanning cost. Next we bound the expected hot pool scanning cost under the assumption of uniformly random edge lengths between 1 and B.

Lemma 21 *If the edge lengths are drawn uniformly at random from $[1, B]$, the expected cost of scanning the hot pools is $O((\mu m \log B)/B)$ I/Os.*

Proof 23 *A category-i edge stored in a hot pool H_j is moved to H_{j+1} if $j < i$ or to H_{j-1} if $j > i$ as soon as a* DeleteMin *operation inspects H_j. Thus, this edge contributes $O(1/B)$ I/Os to the scanning cost of each hot pool H_j with $j \neq i$. Since there are $\log B$ hot pools, the scanning costs of all edges in hot pools not corresponding to their categories is therefore $O((m \log B)/B)$ I/Os. Next we prove that the expected total scanning cost of all category-i edges as part of H_i is $O(\mu m/B)$ I/Os, for any i. Since there are $\log B$ edge categories, the expected total scanning cost of all edges as part of hot pools corresponding to their categories is therefore $O((\mu m \log B)/B)$ I/Os, and the lemma follows.*

First we bound the expected number of category-i edges. The probability that the category of a given edge is i is $(2^i - 2^{i-1})/(B-1) = O(2^i/B)$. Since there are m edges in total, the expected number of category-i edges is therefore $O(m \cdot 2^i/B)$.

Next we bound the number of times a category-i edge xy is scanned while in H_i. The edge cluster E_k containing edge xy is loaded into a hot pool only when the first vertex $z \in V_k$ is visited. Let o_1 be the DeleteMin *operation that returns vertex z, and let o_2 be the* DeleteMin *operation that returns vertex x. Edge xy can be in H_i only between operation o_1 and the first inspection of level i after operation o_2, as this inspection relaxes edge xy. The minimum priority of the vertices returned by operation o_1 is at least $\text{dist}(s, z) - 1$, while o_2 returns x with priority $\text{dist}(s, x)$. Thus, by Lemma 5, H_i is inspected at most*

$2 + 3(\mathrm{dist}(s,x) - \mathrm{dist}(s,z) + 1)/2^{i-1}$ *times while edge* xy *is in* H_i*. Since* H_i *is scanned* $O(1)$ *times as often as it is inspected while* $xy \in H_i$ *and* $\mathrm{dist}(s,x) - \mathrm{dist}(s,z) + 1 \leq \mathrm{dist}(z,x) + 1$*, edge* xy *is scanned* $O(1 + \mathrm{dist}(z,x)/2^i)$ *times in* H_i*. Since* z *and* x *belong to the same vertex cluster* V_k*, there exists a path of at most* μ *edges between* z *and* x*. Each of these edges has length at most* B*. Hence,* $\mathrm{dist}(z,x) \leq \mu B$*, that is, edge* xy *is scanned* $O(\mu B/2^i)$ *times as part of* H_i *and its contribution to the scanning cost of* H_i *is* $O(\mu/2^i)$ *I/Os. By multiplying this with the expected number of category-*i *edges, we obtain the claimed bound of* $O(\mu m/B)$ *I/Os on the expected total cost of scanning category-*i *edges as part of* H_i*.*

By combining this bound on the expected scanning cost of hot pools with the analysis of the cost of the remainder of the algorithm in Section 4.3.4, we obtain a bound of $O(n/\mu + (\mu m \log B)/B + \mathrm{MST}(n,m))$ I/Os on the expected total cost of the algorithm. By choosing $\mu = \sqrt{nB/(m \log B)}$, we obtain that the expected I/O complexity of the algorithm is $O\big(\sqrt{(nm \log B)/B} + \mathrm{MST}(n,m)\big)$ for uniformly random edge lengths between 1 and B.

A.2 Dealing with Short Edges

To prove Theorem 4, we need to add the ability to handle short edges to the algorithm from the previous subsection. These edges create the following problem. In the absence of short edges, the proof of Lemma 13 shows that none of the vertices returned by a DeleteMin operation can be on the shortest path from s to any other vertex returned by the same DeleteMin operation. Thus, the priorities of the returned vertices equal their distances from s. If there are short edges between the retrieved vertices, this may no longer be true. Note, however, that this issue can arise only for vertices that are endpoints of short edges. We call these vertices *special* and all other vertices *regular*. We define the category of a special vertex to be the minimum category of its incident long edges, or 0 if all its incident edges are short. Similarly to the labelling of regular vertices with their categories, the labelling of vertices as special or regular and the computation of the categories of special vertices takes $O(\mathrm{sort}(m))$ I/Os. Now we make the following modifications to the algorithm described in Section A.1.

Again, we form vertex clusters $V_1, V_2, \ldots, V_q$ as in [MM02]'s BFS algorithm, and we form edge clusters $E_1, E_2, \ldots, E_q$ by concatenating the adjacency lists of the vertices in each vertex cluster. However, we do not include any short edges in these clusters. Instead, we collect the short edges in every special vertex's adjacency list E_x in a *special adjacency list* E_x^s. We also store a list L of all special vertices and mark each such vertex as unvisited in L.

After this preprocessing, we run the shortest path algorithm from Section A.1. After a DeleteMin operation returns a set of vertices S, the processing of each vertex in this set depends on whether it is special or regular. Let S_r be the set of regular vertices in S, and S_s the set of special vertices in S. For every regular vertex $x \in S_r$, we insert a Visit$(x, p(x))$ and a Delete(x)

operation into Q_0 as before and output $p(x)$ as the distance from s to x. For the vertices in S_s, we invoke the local shortest path computation described next.

A.2.1 Local Shortest Paths

We start by inserting every vertex $x \in S_s$ into an auxiliary priority queue Q_s, with the priority it had in the bucket heap. We also record x's priority in L. The priority queue Q_s is "normal" in the sense that its DeleteMin operation returns only one element, the one with minimum priority. After initializing Q_s and L in this manner, we run a bounded version of Dijkstra's algorithm on the subgraph of G containing only short edges. As long as Q_s is non-empty, we repeat the following steps: Using a DeleteMin operation, we retrieve the vertex x with minimum priority $p(x)$ from Q_s. We output $p(x)$ as x's distance from s, mark x as visited in L, and insert a Delete(x) and a Visit$(x, p(x))$ operation into bucket Q_0 of the bucket heap. Then we relax the short edges incident to x. To do so, we retrieve x's special adjacency list E_x^s from disk. For every edge xy in E_x^s, we inspect L to determine whether y has already been visited or its current priority $p(y)$ is less than or equal to $p(x) + \ell(xy)$. If one of these conditions holds, we ignore edge xy. Otherwise we distinguish the following two cases.

If $p(x) + \ell(xy) < q_1$, we decrease y's priority in Q_s to $p(y) := p(x) + \ell(xy)$ and store this new priority of y in L. If $y \notin Q_s$, we insert y with priority $p(y)$ instead of decreasing y's priority in Q_s.

If $p(x) + \ell(xy) \geq q_1$, we insert an Update$(y, p(x) + \ell(xy))$ operation into bucket Q_0 of the bucket heap. Note that this has the effect that a special vertex x is visited in the current local shortest path computation only if $p(x) < q_1$. If $p(x) \geq q_1$, visiting x is left to the local shortest path computation of a subsequent iteration.

A.2.2 Correctness

To prove the correctness of the algorithm, we need to prove that every vertex is visited exactly once and that its priority at this time equals its distance from s. First observe that Corollary 3 is unaffected by the addition of a local shortest path computation to the algorithm, that is, a category-i edge xy that has already been loaded into a hot pool reaches H_i by the time a Visit(x, p) operation reaches Q_i unless it has been relaxed already. It is also easy to see that every vertex is visited exactly once, that is, Lemma 11 and, hence, Corollary 4 continue to hold. Indeed, since we handle long edges as in Section A.1, the proof of Lemma 11 shows that the relaxation of a long edge cannot re-insert a vertex into the bucket heap. The relaxation of a short edge xy cannot re-insert vertex y into the bucket heap or Q_s because vertex y is special in this case, that is, its status is recorded in L, and edge xy inserts y into the bucket heap or Q_s only if y is not marked as visited in L. We have the following variant of Lemma 12, where $p_{\max}$ denotes the maximum priority of the elements retrieved from the bucket heap or Q_s so far.

Lemma 22 *Every* UPDATE(y, p') *operation inserted into a bucket* Q_i *satisfies* $q_j \leq p' \leq p_{max} + B$*, where* $j = \max(i, 1)$*. Moreover, this is the last operation in* S_y *and, thus, is correctly recorded as an update of* y*'s priority.*

Proof 24 *If the* UPDATE *is due to the relaxation of a long edge, the proof of Lemma 12 proves the claim. So consider an* UPDATE(y, p') *operation inserted into* Q_0 *during a local shortest path computation. This operation trivially satisfies* $p' \geq q_1$*, because this is the condition for insertion of the operation into* Q_0 *during the local shortest path computation. It satisfies* $p' \leq p_{max} + 1 \leq p_{max} + B$ *because the short edge leading to the insertion of the operation into* Q_0 *has length less than* 1*. It is also the last operation in* S_y *because it is appended to the end of* Q_0*.*

Together with Lemma 11, the following lemma proves the correctness of the algorithm.

Lemma 23 *Every vertex* x *satisfies* $p(x) = \text{dist}(s, x)$ *when it is visited.*

Proof 25 *Again, it suffices to prove that* $p(x) \leq \text{dist}(s, x)$ *because* $p(x) \geq \text{dist}(s, x)$ *follows from all priorities being results of edge relaxations and, thus, representing the lengths of paths from* s *to the corresponding vertices.*

For every vertex $x \neq s$*, let* $\pi_r(s, x)$ *be a shortest path from* s *to* x *such that the predecessor* y *of* x *on this path satisfies* $\text{dist}(s, y) < q_0$ *or* $y = s$*, and let* $\text{dist}_r(s, x)$ *be the length of* $\pi_r(s, x)$*; for* $x = s$*, let* $\text{dist}_r(s, s) = \text{dist}(s, s) = 0$*. (It is intentional that this distance changes as* q_0 *changes.) In order to prove the lemma, we prove two auxiliary claims, which state that (1) every vertex* x *retrieved from the bucket heap is retrieved with priority* $p(x) \leq \text{dist}_r(s, x)$ *and (2) every vertex* x *not visited by the end of the current iteration of the shortest path algorithm satisfies* $\text{dist}(s, x) \geq q_1$*.*

We prove the lemma and the two claims by induction on the iterations of the shortest path algorithm. The first iteration retrieves vertex s *with priority* $p(s) = 0 = \text{dist}(s, s) = \text{dist}_r(s, s)$ *and also visits vertex* s *with this priority. All other vertices visited during the first iteration have distance less than* 1 *from* s*. Thus, the shortest paths from* s *to these vertices consist only of short edges. Since we run Dijkstra's algorithm on the short edges without modifications, each such vertex* x *is visited with priority* $p(x) = \text{dist}(s, x)$*. A vertex* x *not visited during the first iteration satisfies* $\text{dist}(s, x) \geq 1 = q_1$ *because the local shortest path computation terminates only when there are no more vertices left in* Q_s*, and every special vertex* x *with* $\text{dist}(s, x) < q_1$ *is inserted into* Q_s *by its predecessor on the shortest path* $\pi(s, x)$ *from* s *to* x*. Hence, the lemma and the two claims hold for the first iteration.*

So consider a vertex x *that is visited during a subsequent iteration and assume the lemma holds for all previous iterations. If vertex* x *is retrieved from the bucket heap by the current* DELETEMIN *operation, then its priority is easily seen to be at most* $\text{dist}_r(s, x)$*. Indeed, by the induction hypothesis, its predecessor* y *in* $\pi_r(s, x)$ *is visited during a previous iteration with priority* $p(y) = \text{dist}(s, y)$*. If edge* yx *is long, the proof of Lemma 13 shows that*

this edge is relaxed before vertex x can be retrieved with a priority greater than $\text{dist}(s,y) + \ell(yx) = \text{dist}_r(s,x)$*. If edge yx is short, it is relaxed immediately when vertex y is visited, leading to an immediate update of $p(x)$ to a value no greater than* $\text{dist}(s,y)+\ell(yx) = \text{dist}_r(s,x)$*. This proves claim (1) for the current iteration.*

Next we prove claim (2), that is, that no vertex at distance less than q_1 from s is visited after the current iteration. Consider such a vertex x.

If $\text{dist}_r(s,x) = \text{dist}(s,x) < q_1$*, then its predecessor y on $\pi_r(s,x)$ is visited in a previous iteration. If edge yx is relaxed during or before the current iteration, then* $p(x) \leq \text{dist}_r(s,x) < q_1$*, and vertex x is retrieved in the current iteration. If edge yx is relaxed after the current iteration, it must be a long edge and the first inspection of the level i corresponding to its category after visiting vertex y happens after the current iteration. By Lemma 22, however, the* UPDATE(x,p) *operation inserted into Q_i when edge yx is relaxed satisfies* $p = \text{dist}(s,y) + \ell(yx) = \text{dist}_r(s,x) \geq q_i \geq q_1$*, a contradiction. Thus, edge yx must be relaxed during or before the current iteration, and vertex x is retrieved from the bucket heap in the current iteration. If vertex x is regular, this implies that the current iteration visits vertex x. Otherwise x is inserted into Q_s during the initialization of the local shortest path computation, with a priority less than q_1, which guarantees that the local shortest path computation of the current iteration visits x.*

If $\text{dist}(s,x) < q_1$ *and* $\text{dist}(s,x) < \text{dist}_r(s,x)$*, then $\pi(s,x)$ consists of a path* $\pi(s,y) = \pi_r(s,y)$ *to a vertex* $y \in \pi(s,x)$ *satisfying* $q_0 \leq \text{dist}(s,y) < \text{dist}(s,x) < q_1$*, followed by a shortest path $\pi(y,x)$ from y to x. As just argued, vertex y is retrieved from the bucket heap in the current iteration and visited with priority* $p(y) = \text{dist}_r(s,y) = \text{dist}(s,y)$*. Since* $\text{dist}(y,x) = \text{dist}(s,x) - \text{dist}(s,y) < q_1 - q_0 \leq 1$*, $\pi(y,x)$ consists of only short edges, and vertex x is visited with priority* $p(x) \leq \text{dist}(s,y) + \text{dist}(y,x) = \text{dist}(s,x)$ *during the current local shortest path computation.*

Since this already proves that every special vertex is visited with $p(x) = \text{dist}(s,x)$ *during the local shortest path computation, it suffices to prove that every regular vertex is visited with* $p(x) = \text{dist}(s,x)$*. However, for a regular vertex visited in the current iteration,* $\text{dist}(s,x) = \text{dist}_r(s,x)$*. Since we have just proven that vertex x is retrieved with priority* $p(x) \leq \text{dist}_r(s,x)$ *from the bucket heap, this implies that* $p(x) = \text{dist}(s,x)$ *at the time vertex x is visited.*

A.2.3 I/O Complexity

The local shortest path computation cannot affect the cost of manipulating priority queue buckets and hot pools. Since Corollary 4 continues to hold, the expected cost of the algorithm excluding the cost of local shortest path computations is therefore $\text{O}\big(\sqrt{(nm\log B)/B} + \text{MST}(n,m)\big)$ I/Os, as argued in Section A.1. Next we bound the cost of the local shortest path computations. We start by bounding the expected number of short edges and special vertices.

Observation 3 *The expected number of short edges is m/B. The expected number of special vertices is at most $2m/B$.*

Proof 26 *Since edge lengths are drawn uniformly at random from the interval $(0, 1]$ and an edge is short if its length is less than $1/B$, the probability that an edge is short is $1/B$, leading to an expected number of m/B short edges. Since every short edge makes only its two endpoints special, the expected number of special vertices is at most $2m/B$.*

Lemma 24 *The expected cost of the local shortest path computations is $O(m/B+$ $\mathrm{sort}(m/B))$ I/Os.*

Proof 27 *We perform $O(m/B)$ random accesses into L, one when each special vertex is retrieved from the bucket heap and $O(1)$ per short edge. The number of random accesses to special adjacency lists is bounded by the number of special vertices, which is $O(m/B)$. Since these lists have expected size $2m/B$, the cost of scanning them is $O(m/B^2)$ I/Os.*

It remains to analyze the cost of priority queue operations. Clearly, the number of these operations is $O(m_s)$, where m_s is the number of short edges. If we use the buffer tree by [Arg03] to implement Q_s, these operations cost $O(\mathrm{sort}(m_s))$ I/Os, which is $O(\mathrm{sort}(m/B))$ I/Os in expectation because $E[m_s] = m/B$. Note that the buffer tree does not support Update *operations. While such an operation can be simulated by first deleting the vertex and then re-inserting it with the updated priority, deletions on the buffer tree require the current priority of the deleted element as an argument, which is what prevents this strategy from being used as a general replacement for an* Update *operation. In our case, however, we access L before performing an* Update *on a vertex in Q_s and, thus, know its current priority. Thus, we can simulate* Update *operations using a deletion followed by an insertion.*

Summing the costs of the different parts of the local shortest path computation, we obtain the bound stated in the lemma.

As argued above, the cost of the algorithm excluding the cost of the local shortest path computations is $O\big(\sqrt{(nm \log B)/B} + \mathrm{MST}(n, m)\big)$ I/Os and, by Lemma 24, the cost of the local shortest path computations is dominated by this. The correctness of the algorithm is established by Lemmas 11 and 23. Thus, Theorem 4 is proved.

B A Faster Algorithm for Random Edge Lengths

Our final result shows how to obtain a faster average-case efficient algorithm than in Section A. The algorithm is the algorithm from Section 4, except that we deal with special vertices and short edges as described in Section A.2. The correctness of this algorithm follows from the correctness of the algorithm in Section 4 and the arguments in Section A.2.2. Next we analyze its expected I/O complexity.

Theorem 5 *The single-source shortest path problem on an undirected graph with n vertices, m edges, and uniformly random edge lengths in the interval $(0,1]$ can be solved using expected $\mathrm{O}\Big(\sqrt{nm/B} + (m/B)\log B + \mathrm{MST}(n,m)\Big)$ I/Os.*

Proof 28 *We divide the algorithm into three parts: (1) the algorithm from Section 4 excluding the scanning of hot pools and tree buffers, (2) the scanning of hot pools and tree buffers, and (3) local shortest path computations. Part (1) is independent of the edge lengths and has cost $\mathrm{O}(n/\mu+(m/B)\log B+\mathrm{MST}(n,m))$ I/Os, as argued in Section 4.3.4. The expected cost of part (3) is $\mathrm{O}(\mathrm{sort}(m))$ I/Os, as shown in Section A.2.3. Note that the changed placement of long edges into hot pools compared to the algorithm from Section A has no impact on the local shortest path computations. Next we prove that the expected cost of part (2) is $\mathrm{O}((m/B)\log B + \mu m/B)$ I/Os.*

First consider the cost of scanning hot pools. By Lemma 18, every category-i edge contributes $\mathrm{O}(\log B/B)$ I/Os to the scanning cost of hot pools $H_1, H_2, \ldots, H_{i-1}$ and $\mathrm{O}(\mu(r-i+1)/B)$ I/Os to the scanning cost of hot pools $H_i, H_{i+1}, \ldots, H_r$. Hence, the cost per category-i edge is $\mathrm{O}((\log B + \mu(\log B - i))/B)$ I/Os. The expected number of category-i edges is $\mathrm{O}\big(2^i m/B\big) = \mathrm{O}\big(m/2^{\log B - i}\big)$. If m_i is the actual number of category-i edges, the expected scanning cost of category-i edges is thus $\mathrm{O}\big((m_i/B)\log B + \big(\mu m/(2^{\log B-i}B)\big)(\log B - i)\big)$ I/Os. By summing over all categories, we obtain an expected hot pool scanning cost of

$$\sum_{i=1}^{r} \mathrm{O}\Big(\frac{m_i}{B}\log B + \frac{\mu m}{2^{\log B-i}B}(\log B - i)\Big) = \mathrm{O}\left(\frac{m}{B}\log B\right) + \mathrm{O}\left(\frac{\mu m}{B}\right)\sum_{i=1}^{r}\frac{\log B - i}{2^{\log B - i}}$$
$$\leq \mathrm{O}\left(\frac{m}{B}\log B\right) + \mathrm{O}\left(\frac{\mu m}{B}\right)\sum_{i=0}^{\infty}\frac{i}{2^i} = \mathrm{O}\left(\frac{m}{B}\log B + \frac{\mu m}{B}\right) \text{ I/Os.}$$

To bound the expect tree buffer scanning cost, we bound the scanning costs of cluster tree roots, non-root non-leaf cluster tree nodes, and cluster tree leaves separately. By Lemma 16, the scanning cost of cluster tree roots is $\mathrm{O}((m/B)\log B)$ I/Os. By Corollary 5, every non-root non-leaf cluster tree node $[x]_i$ contributes $\mathrm{O}((\log B + \mu(\log B - i))/B)$ I/Os to the tree buffer scanning cost. Each such cluster tree node $[x]_i$ is a minimal i-component and, hence, contains at least one category-i edge. Since the minimal i-components are edge-disjoint, this implies that the expected number of minimal i-components is bounded by the expected number of category-i edges, and the argument from the previous paragraph bounds the contribution of non-root non-leaf cluster tree nodes to the tree buffer scanning cost by $\mathrm{O}((m/B)\log B + \mu m/B)$ I/Os.

Finally consider cluster tree leaves. It suffices to consider only leaves that are not also roots of their cluster trees, as we have already accounted for the cost of scanning cluster tree roots. It is not hard to see that, while such a leaf $[x]_0$ is stored in a tree buffer R_j, either its parent $[x]_i$ in $\mathcal{C}(G)$ is stored in the same tree buffer R_j or Q_j contains an $\textsc{Update}(x,p)$ operation with $p < q_{j+1}$.

We call an access to $[x]_0$ while the first condition is true a type-I access, and an access to $[x]_0$ while the second condition is true a type-II access.

The cost of type-I accesses to $[x]_0$ is bounded by the scanning cost of $[x]_i$. Thus, by Lemma 16, the total cost of type-I accesses to all leaves whose parents are cluster tree roots is $O((m/B)\log B)$ I/Os. Similarly, if $[x]_i$ is not a cluster tree root, then, by Corollary 5, the cost of type-I accesses to $[x]_0$ is $O((\log B + \mu(\log B - i))/B)$ I/Os. Every cluster tree leaf $[x]_0$ whose parent is an i-component $[x]_i$ is an endpoint of a category-i edge. Thus, the number of such cluster tree leaves is at most twice the number of category-i edges, and the same argument as above bounds the expected total cost of type-I accesses to cluster tree leaves with non-root parents by $O((m/B)\log B + \mu m/B)$ I/Os.

The cost of each type-II access to $[x]_0$ in H_j can be charged to an access to the corresponding UPDATE(x, p) *operation in Q_j. Thus, since the total cost of priority queue operations is $O(\mathrm{sort}(m) + (m/B)\log B)$ I/Os (Lemma 14), the total cost of all type-II accesses to cluster tree leaves is $O((m/B)\log B)$ I/Os (the $O(\mathrm{sort}(m))$ I/Os in the cost of priority queue operations account for sorting and scanning Q_0, which does not correspond to type-II accesses to cluster tree leaves).*

By summing the costs of the three parts of the algorithm, we obtain an expected cost of $O(n/\mu + (\mu m/B) + (m/B)\log B + \mathrm{MST}(n, m))$ I/Os for the entire algorithm. The first two terms are balanced if we choose $\mu = \sqrt{nB/m}$, which leads to the I/O complexity claimed in the theorem.

Note that, for sparse graphs (i.e., graphs with $m = O(n)$), the first term in the bound in Theorem 5 dominates the second one, and the complexity of the algorithm becomes $O\Big(n/\sqrt{B} + \mathrm{MST}(n, n)\Big)$ I/Os, which matches the performance of [MM02]'s BFS algorithm up to constant factors.